UNIVERSITY OF LONDON THESIS

Degree PHD Year 2008 Name of Author DE OLIVEIRA, JOAO, FERNANDO, DOS SANTOS, FRADINHO, DUARTE.

Vertex classification for non-uniform geometry reduction

João Fernando dos Santos Fradinho Duarte de Oliveira

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

29th July 2008

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

I, João Fernando dos Santos Fradinho Duarte de Oliveira, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm this has been indicated in the thesis.

Abstract

Complex models created from isosurface extraction or CAD and highly accurate 3D models produced from high-resolution scanners are useful, for example, for medical simulation, Virtual Reality and entertainment. Often models in general require some sort of manual editing before they can be incorporated in a walkthrough, simulation, computer game or movie. The visualization challenges of a 3D editing tool may be regarded as similar to that of those of other applications that include an element of visualization such as Virtual Reality. However the rendering interaction requirements of each of these applications varies according to their purpose. For rendering photo-realistic images in movies computer farms can render uninterrupted for weeks, a 3D editing tool requires fast access to a model's fine data. In Virtual Reality rendering acceleration techniques such as level of detail can temporarily render parts of a scene with alternative lower complexity versions in order to meet a frame rate tolerable for the user. These alternative versions can be dynamic increments of complexity or static models that were uniformly simplified across the model by minimizing some cost function. Scanners typically have a fixed sampling rate for the entire model being scanned, and therefore may generate large amounts of data in areas not of much interest or that contribute little to the application at hand. It is therefore desirable to simplify such models non-uniformly.

Features such as very high curvature areas or borders can be detected automatically and simplified differently to other areas without any interaction or visualization. However a problem arises when one wishes to manually select features of interest in the original model to preserve and create stand alone, non-uniformly reduced versions of large models, for example for medical simulation. To inspect and view such models the memory requirements of LoD representations can be prohibitive and prevent storage of a model in main memory. Furthermore, although asynchronous rendering of a base simplified model ensures a frame rate tolerable to the user whilst detail is paged, no guarantees can be made that what the user is selecting is at the original resolution of the model or of an appropriate LoD owing to disk lag or the complexity of a particular view selected by the user. This thesis presents an interactive method in the context of a 3D editing application for feature selection from any model that fits in main memory.

We present a new compression/decompression of triangle normals and colour technique which does not require dedicated hardware that allows for 87.4% memory reduction and allows larger models to fit in main memory with at most 1.3/2.5 degrees of error on triangle normals and to be viewed interactively. To address scale and available hardware resources, we reference a hierarchy of volumes of different sizes. The distances of the volumes at each level of the hierarchy to the intersection point of the line of sight with the model are calculated and these distances sorted. At startup an appropriate level of the tree is automatically chosen by separating the time required for rendering from that required for sorting and constraining the latter according to the resources available. A clustered navigation skin and depth buffer strategy allows for the interactive visualisation of models of any size, ensuring that triangles from the closest volumes are rendered over the navigation skin even when the clustered skin may be closer to the viewer than the original model. We show results with scanned models, CAD, textured models and an isosurface.

This thesis addresses numerical issues arising from the optimisation of cost functions in LoD algorithms and presents a semi-automatic solution for selection of the threshold on the condition number of the matrix to be inverted for optimal placement of the new vertex created by an edge collapse. We show that the units in which a model is expressed may inadvertently affect the condition of these matrices, hence affecting the evaluation of different LoD methods with different solvers. We use the same solver with an automatically calibrated threshold to evaluate different uniform geometry reduction techniques. We then present a framework for non-uniform reduction of regular scanned models that can be used in conjunction with a variety of LoD algorithms. The benefits of non-uniform reduction are presented in the context of an animation system.

Most high quality geometry reduction methods make two important assumptions about the input surface. The first assumption is that the surface has no duplicate vertices that under simplification would lead to cracks and holes. The second is that the surface is consistently oriented. Often neither assumption holds and there are many 3D models obtained from sources that are no longer available or that have been produced by use of different modelling packages that cannot therefore be simplified adequately. We present two solutions for automatically solving these problems.

Acknowledgements

I gratefully acknowledge support from the following funding agencies without which I could not have taken up this Ph.D. research project.

- Fundação Calouste Gulbenkian, Serviço de Educação e Bolsas da Fundação Calouste Gulbenkian Av. de Berna N. 45A 1067 - 001 Lisboa Portugal
- JNICT/PRAXIS XXI/BD/15530/97 Fundação para a Ciência e a Tecnologia Av. D. Carlos I, 126, Piso C 1200 Lisboa Portugal
- In the appendix of this thesis, the contribution of co-author Dongliang Zhang's was funded by EPSRC grant reference GR/M46082/01.

I would also like to thank Bernard Buxton, Stuart Robson, Marek Ziebart, Matt Hall, and Anthony Sibthorpe for comments on earlier drafts.

I thank Alf Linney for producing models with his face scanner, Hamamatsu Photonics UK for the loan of the Body Line Scanner to the Department of Computer Science. This scanner was used to obtain several of the whole body scans used as examples, in particular for animation in appendix. I thank Ciara, Jorge and volunteers also for the creation of the body scans.

We would like to acknowledge Matti Hämäläinen & Bruce Fischl for providing the brain model and Simon Arridge & Athanasios Zacharopoulos for originating this project, Peter Lindstrom and Martin Isenberg for providing stream data, W. B. Langdon for advice on pseudo-random numbers and Gideon Amos, Miles Hansard and Jesper Mortensen for discussions on C++ software optimisation.

A number of models were reported in this thesis which we would like to acknowledge:

The Digital Michelangelo Project, Stanford University for the scanned model of Michelangelo's statue of David, Stanford University and Marc Levoy for the bunny, Lucy, Thai, Happy buddha and dragon model, Geometrix for the dragon model with a "flat" base and the model of the monster with a club, As-Built Solutions for the power plant model, UNC for the massive power plant model, GE Aircraft Engines for the turbine blade model, Richard Marsden for the

mobius strip model, Bernhard Spanlang for the manequin model. Hugues Hoppe for the Cessna aeroplane CAD model. Michael Garland for the Crater Lake model. SGI/Powerflip for the cow model. Tamal Dey for the scanned dinossaur model. Stuart Robson/British Museum, The Scan Team/PCA for the canoptic chest dataset. Arius 3D for the Skull scan.

Instituto Português do Partrimónio Arquitectónico and Artescan for the scan of the Batalha cathedral.

Finally I would like to thank my parents, my brothers Jorge and Manuel, Ben Dias, Ana Franco, Francis Gioia, Escola Salesiana do Estoril, Dona Antónia, the Muchaxo establishment, Lisa, and last but not least Marion for unwaivering support.

Contents

List of Algorithms

List of Tables

List of Figures

Chapter 1

Introduction

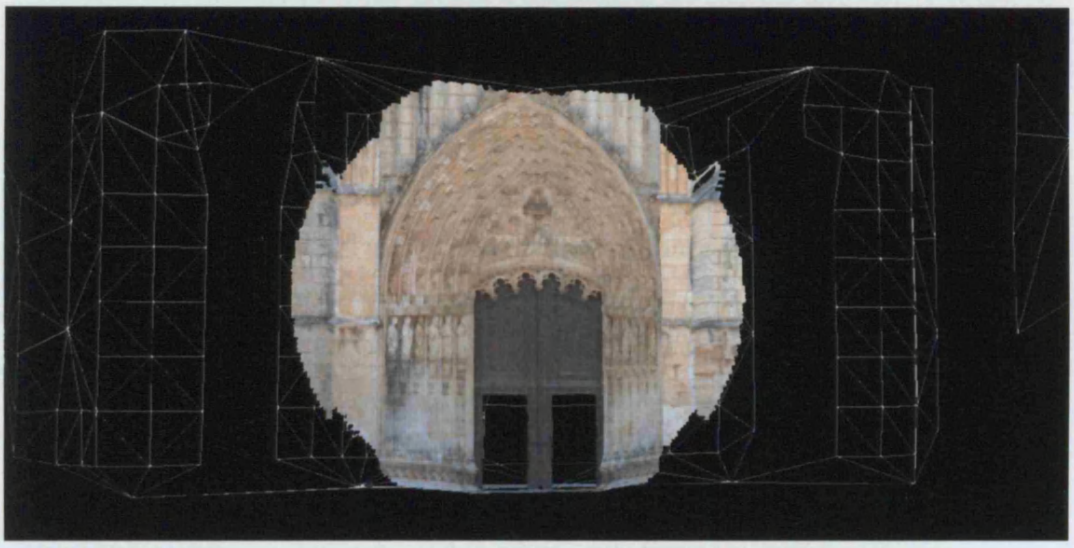


Figure 1.1: Mosteiro da Batalha, Interactive large model visualization with developed system.

3D models in Computer Graphics are increasingly used to aid the task of modelling and measuring the real world through a growing number of applications. These include large-scale applications such as: industrial surveying/scanning of dams, tunnels, rail tracks and the space shuttle; small-scale applications such as: the simulation of facial surgery with finite element meshes [KGC+96] and visualization and modelling of internal organs and bones from medical derived data [LDS03]; and computer-aided applications such as: air-flow simulation with (CAD) models; Global Information Systems (GIS) [G.90]; and systems for the preservation and restoration of historical artefacts [LPC+00, RCC+01].

These 3D models can be represented by means of a set of discrete primitives: polygons, such as triangles, voxels or tetrahedral volume grids [CM02], and points [RL00]. Alternatively, impostor images [TLC02] may be used or intermediate representations such as the parametric representation of a subdivision surface or BSpline curved surface that is later projected and

rendered by a rendering system as a collection of triangles to within a specified pixel tolerance. Owing to the simple geometry of triangles and the efficiency with which they may be rendered, triangles have enabled the development of graphics hardware at personal consumer level capable at the time of writing of rendering millions of triangles per second [Coh99].

In terms of the number of geometric primitives or size of 3D models we identify three categories. The first category consists of models that do not fit in main random access memory (RAM) and use rendering system solutions that access secondary memory to visualize the model which resides out-of-core [CRMS03]. In the second category there are models that are small and trivially rendered that fit in RAM whilst finally, in the third category we put models that fit in core memory but either present a challenge to the graphics hardware or cannot be displayed at interactive rates.

For several years, a lot of research was focused on transforming models from the third category into smaller models of the second category. Such level of detail (*LoD*) techniques [SZL92, RB93, HDD+93, KT96, Gue96, CVM+96, GS97, LT99] created smaller discrete approximations to a model for faster rendering. Later, viewing parameter methods such as frustrum culling [Cla76], silhouete preservation and finer detail selection within regions foveated by the viewer were incorporated into dynamic rendering systems [Hop96, XV96, LE97].

In these, the history or sequence of individual geometric simplification operations of a model would be used to build a *hierarchical vertex tree* that would allow an active processing front within the tree to combine parts of the original geometry with simplified geometry to speed-up the rendering. Research was then subsequently focused on models of the first category and, in particular, methods for reducing these models into smaller models of the second category or third were developed [Lin00, LS01a, FCGW02, GS02, CRMS03]. Finally, view-dependent systems were developed for visualizing models in the first category of complexity [DR02]. Decoupling of out-of-core fetches by means of asynchronous rendering was introduced for producing high frame rates at anytime or at any viewing parameters by means of whatever LoDs happened to be present in the graphics card [DR02, CGG+04, YSGM04, BGB+05, GM05].

A noticeable trend is that the models that were deemed out of core in the past are migrating to in-core today. An example of such a model is the scanned statue of Lucy comprised of 28 million triangles for which, with some compression of attributes as developed in this thesis, all triangles are directly renderable and reside within 739 MBytes of RAM. With the advent of 64-bit architectures, main memory is no longer limited to the 4 Gigabytes (4×10^9bytes) restriction of 32-bit machines and the theoretically accessible RAM is 16 billion, billion bytes (16×10^{18}bytes), 16 hexabytes, or 16 million terabytes, where 1 Terabyte is 1024 Gigabytes.

At present some 64-bit personal computers such as Apple's G5 [app06] have restricted the addressable RAM to 2^{42}bits, or 4 Terabytes and provide personal computer models with up to 8 Gbytes of physical main memory. In spite of such advances in address space and RAM capacity, it is not inconceivable that there will be always models from the first category. However, it is also not inconceivable that new applications with mobile devices such as phones and personal digital assistants (PDAs) which do not have secondary memory would benefit from technology developed to cater for models from the second and third categories. The principles demonstrated in this thesis are aimed at dealing with models in the second and third category but can also be used for models of the first category. We also note that whilst some models might themselves fit in main memory, the total size of their LoD data structures forces them in many applications to be in the first category. In contrast, the visualization system presented in this thesis (Figure 1.1) has no such memory overhead thereby allowing such models to remain in the third category. Similarly, thanks to our attribute compression algorithm [OB06], some models that from their raw size alone belong intrinsically to the first category also remain in the third category. In the next section we present the problem addressed by this thesis. In Section 1.2 we present the aims we set out to address and in Section 1.3 we state the main hypothesis of this thesis. In Section 1.4 we present a list of contributions, in Section 1.5 we list publications produced and, finally, in Section 1.6 we present a road-map to the content of this thesis.

1.1 The Problem

Most geometry reduction techniques design an error cost function, such as volume error [Gue96] or distance error to a set of planes [GS97], and assign a potential error or cost to every geometric primitive such as an edge that might be simplified or deleted. A greedy algorithm then proceeds to apply simplification operations to the smallest potential cost elements, independent of location and of updating costs in affected surrounding areas. New vertex positions and feature attributes that are optimal with respect to the chosen error function can be found in order to approximate the original object. Some methods are driven through a target global error [HDD+93, CVM+96] criterion rather than a target polygon budget. There has been some debate [OB01, LT99] as to whether versions of a model obtained from approximated, optimised vertex positions versions are necessarily better than versions that use only vertex positions included in the original model. In the context of a 3D editing tool, a computer graphics modeller confronted with the task of creating models with a specific polygon budget will want a general, non-uniform reduction in the level of detail of a model. For this purpose, the modeller will typically manually select features and use the automated, uniform reduction

facilities of the tool only in parts of the reduction process as a whole [Ste98]. Unfortunately, not many level of detail strategies allow for the integration of uniform and non-uniform reduction. Some automatic methods [SZL92, GS97, LT99] can detect border vertices and simplify them in a different manner to the rest of the model. For example, features such as high curvature or borders can be detected automatically without visualization by an offline process through their connectivity information alone. Solutions available for general non-uniform geometry reduction [KG03, LW01] often make the assumption that one has the hardware capabilities to be able to interact with the original model. Level of detail solutions have been created to accelerate the rendering by reducing the number of primitives according to a uniform criterion. However, when one wishes to reduce the level of detail non-uniformly an interesting question arises; namely that one needs to work, interact with and reason about the original *zero pixel error* resolution of a complex model from *any viewing angle.* This raises questions about *interactivity* and *modelling*.

In principle, one could resort to incorporating LoD for viewing 3D models in a 3D editing tool by uniformly simplifing a model in main memory or secondary memory and use LoD switching to aid the interaction with the model we wish to mark for non-uniform reduction. However updating the active front of vertex tree datastructures in gaze directed rendering systems such as [LHNW00, WWHW96] can create rendering lags during large view changes with massive models. Unlike Virtual Reality applications where the user can tolerate rendering content lags while finer resolution data is paged-in, in a model editing task it is desirable always to have present for immediate reasoning and productivity the finest resolution data of the area at which the user is looking. The issue of lags has been somewhat alleviated by allowing complete surfaces to be switched by complete surface LoDs without waiting for updates at the triangle level [BGB+05] by means of asynchronous rendering and Erikson et al.'s [EM00] hiearchical LoDs (HLoDs) for CAD objects. Whilst asynchronous rendering has eliminated system lags where the user would have to wait for the system to render frames in order to continue viewing interaction, the rendering content lag associated with the update of vertex trees or with paging-in geometry from disk remains. These systems can effectively shift the computational burden of system lags to rendering content lags less perceptible to the user; i.e. frame rates are maintained, but the user still has to wait for fine resolution geometry to be paged-in and rendered. In addition, it is also difficult to evaluate the performance of such asynchronous systems whereas the rendering performances reported in this thesis are for synchronous renderings including system time and their performance is easy to evaluate.

It can be seen in Figure 1.2 that if during editing the user makes rapid repeated view

changes from PA to PB in, for example, a shape proportion reasoning task, the following modelling or interactivity limitations become apparent:

1. Whilst surface patches M1 and N1 might be stored permanently in the graphics card memory for quick retrieval [BGB+05], fine resolution surfaces M4 and N4 will still have a lag associated when paging-in. In the context of an editing task where repeated viewing of the same view points is often necessary these lags are not desireable, at least for models that fit in main memory. (*an interactivity limitation*)

2. The patch size B in terms of number of triangles, represented in the figure by the horizontal line below N4 on the right, is chosen and fixed with the performance of a target platform in mind making the solution less adaptable when running on other systems. Again, in the context of a 3D editing tool such software is expected to run on a variety of different systems with different capabilities. (*another interactivity limitation*)

3. In the case of textured models manual intervention is required to align textures to suitable size data blocks [BGB+05]. In the context of a 3D editing tool, however, it is desirable that any model loaded for viewing is processed automatically and rendered. (*a modelling limitation*)

4. Whilst detail is being paged-in the user unknowingly might be incorrectly selecting triangles that are not at the finest resolution. (*a modelling limitation*)

5. For 3D models whose surface patches such as M4 and N4 fit in main memory the required memory for storing M3, M2, M1, N3, N2, N1 might dictate the use of an inherently slower out-of-core visualization system. (*another modelling limitation*)

6. One feature expected from any 3D editing tool is the ability to rapidly load/import browse between several models. The pre-processing time for the construction of LoDs of large models, such as the 56 million triangle model of David, can take over 1 hour[1] with triangle based visualization systems [BGB+05, CGG+04] but of the order of 12 minutes[2] with QSplat [RL00] a similar specified point based systems. Current industry solutions which do not use LoD, alternatively skip rendering triangles according to a set ratio (e.g. 1/64) during interaction, this solution is clearly not ideal when editing/reasoning with complex models such as an isosurface. (*a usability limitation*)

[1] scaling published results of multiple PC CPUs running at above 1.6 GHz.

[2] scaling published results of a SGI Onyx2 Infinity Reality workstation.

We henceforth refer to the above limitations as the six main limitations on the use of level of detail techniques in a 3D editing tool. The main problem tacked by this thesis work is to build a 3D editing tool that, rapidly and without manual intervention, allows for immediate access to fine detail at interactive rates for models comprised of large numbers of triangles whether they are textured or not, provided that they fit in main memory. Secondly, the work described in this thesis addresses the geometric and numerical issues whose resolution is necessary in order to make possible the uniform or non-uniform geometry reduction of a broad range of models.

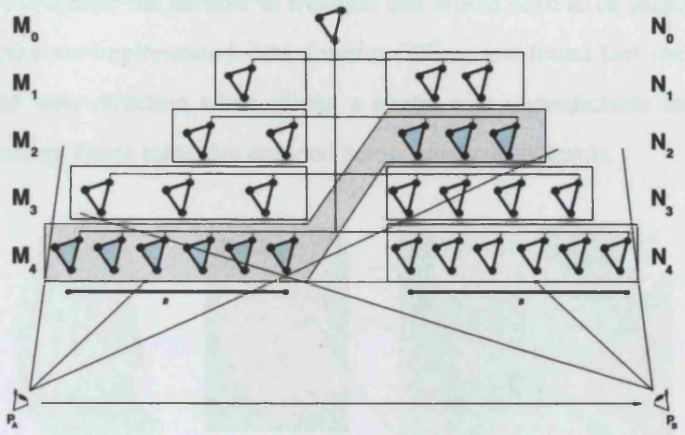


Figure 1.2: Interactivity and modelling issues in a LoD system. Highlighted in blue is the *active front* comprised of zero pixel error surface patch M4 and LoD patch N2, being rendered for the view from PA. The data being rendered from PB will temporarily be that of PA.

1.2 Aims

The main aims of this thesis arose when two colleagues from the Medical Imaging group of the Department of Computer Science at University College London approached me with the following challenge:

I was given a ~600,000 triangle isosurface[3] model (a model of size category 3 for most laptops) of the grey and white matter of the left side of the human brain[4]. A full, symmetrical model of the brain was required as a triangulated surface as the starting point for several other parameterised and smoother representations to be utilised in an optical tomography system (Zacharoupoulos et al. [ZSA04]). An extensive edit of both the white and grey matter

[3] A surface computed for visualization purposes derived from similar property volume data.

[4] The model of the brain used in this thesis was created using a surface reconstruction tool called FreeSurfer (see Fischl et al. [FSD99]) to extract the isosurface from an MRI scan. The scan was acquired using 2 MP-RAGE scans (8.5 minutes/scan), motion corrected and averaged, collected on a Siemens 1.5T Sonata machine.

isosurfaces was thus required in which approximately 25,000 triangles had to be selected and deleted in order to extrude the deleted contour to the mid-brain vertical plane and create a connecting but elsewhere non-intersecting symmetrical, mirror-image right hand side to the given left-side brain model. The boundaries of the two surface areas to be deleted can be seen in red and blue in the rightmost image of Figure 1.3. The grey and white matter surfaces prior to mirroring were together comprised of 596,872 triangles. The resulting symmetrical model had approximately 1.1 million triangles.

In an effort to reduce the number of triangles that would have to be rendered for guiding the editing process we implemented view frustum culling, but found that the user's changes of viewpoint and view direction when editing a model were unpredictable and consequently caused discomforting frame rates that dropped below interactivity levels.

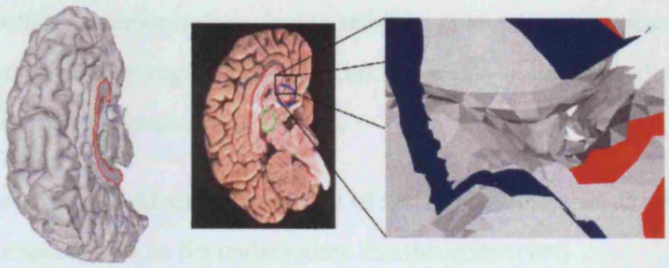


Figure 1.3: Highlighted features for editing the grey and white matter isosurface model of the left hemisphere of the human brain; *left:* 596,872 triangle model created from isosurface extraction; *middle:* anatomical photo for guiding the edit; *right:* close up of marked areas for removal.

In order to carry out this editing task, and similar tasks in a more general setting, it is desirable to have a system that would overcome the six limitations outlined in the previous section, namely:

1. One that would allow *immediate* viewing without lags of the parts of a model at which the user is gazing, at the finest resolution geometry, at interactive rates, from any viewing direction, and simultaneously providing granular control over the extent of the fine resolution geometry that is displayed. This would aid the user selection, reasoning about and editing of different areas for potential deletion or non-uniform level of detail reduction.

2. One that could adapt to the available processing resources and provide, at run-time, control over the rendering speed in a manner independent of the size of the model in memory and of the specific capabilities or availability of a graphics card.

3. One that would allow interactive viewing and would support intrinsically different models without any manual, individual pre-processing, whether the models be obtained from scanned or CAD data, by isosurface extraction, and whether they are textured or not.

4. One that ensures that what the user selects/clicks in the foveated area is always of the finest resolution of the model and not for example of a reduced LoD.

5. One that would not impose significant memory overheads on models and would, if possible, reduce their overall memory requirement so as to allow for the visualization of models that would otherwise not fit in main memory.

6. One that, given editing tasks of the size and complexity of that requested for the brain model together with the difficult non-self intersection constraints that the extrusions of the grey and white matter isosurfaces had to satisfy, would deliver a continuously high frame rate from any viewing angle. This is crucial for the physiological well being [WSNR96] of the editor in hour-long editing sessions.

We refer to the above outlined six requirements of the system as our six Primary aims with the following three requirements as Secondary aims that the system be:

1. One that would provide general non-uniform reduction heuristics that could be applicable to a variety of different popular error functions so as to support/combine their different characteristics, and that would preserve borders and colour discontinuities.

2. One that would detect and delete duplicate vertex geometry in memory so as to enable cohesive surface reduction.

3. One that would fix automatically surface normal inconsistencies.

The main aim of this thesis was to develop a method and system that would enable the manual marking of feature areas on large models for the purposes of non-uniform reduction. In summary, simply put, the aim of this thesis was to create a 3D tool, with or without uniform Level of Detail, that would allow one to interact with and edit a broad range of large models.

1.3 Hypothesis Statement

In this section we state the two main hypotheses of this thesis. In order to be able to test the second hypothesis we need to address the first hypothesis. Specific terms and phrases used within each hypothesis are clarified thereafter.

1.3.1 Main Hypothesis Statement #1

"It is possible, without the assumption that the model is small enough for interaction, to develop a system that enables one to interact with and designate 3D features on a model stored in main memory, for example, for non-uniform geometry reduction."

- **"small enough for interaction":** Existing non-uniform geometry reduction techniques that support manual selection of features assume that the model can be rendered comfortably and hence do not use rendering acceleration techniques such as uniform level of detail to enable the interaction. Using LoD techniques could compromise the selection of original features and hence is to be avoided. Frequently the literature refers to minimum display rates for interaction as 5 frames per second and high frame rates as being over 15-25 fps. Here, we envisage a system that can, without graphics acceleration hardware, provide minimal interaction for any size model residing in main memory and high frame rates if provided with a comodity graphics acceleration card.

1.3.1.1 Hypothesis #1.1

"Such a system can provide and maintain high frame rates with immediate, uncluttered rendering of the finest resolution data of a model in the user's foveated direction of view with flexible run-time control over the extent of that data and rendering speed independently of the viewing direction, the size of the model in main memory, the use of specific graphics acceleration hardware, and whether the model is textured, derived from scanned data, an isosurface, or from CAD, etc."

- **"immediate":** Recent visualization systems provide asynchronous rendering which has enabled high frame rates of what is in a graphics card memory at anytime. However, for such systems, there is a lag in updating that memory with the fine detail data of what the user is currently foveating. Immediate here means that there is no delay of this kind.

- **"uncluttered rendering":** This means that the data being rendered in the user's foveated direction of view is guaranteed to be that of the finest detail of the model and not some LoD that would hinder the task of model selection.

- **"finest resolution data":** This means unaltered original geometry.

- **"provide and maintain high frame rates":** Here we refer to synchronous rendering according to the user's chosen viewing parameters and usually, at any time or viewing condition, a minimum frame rate approximately above 15 frames per second.

- **"flexible run-time control over the extent of that data and rendering speed":** This means that the user can at run-time trade computation time with rendering time to further increase the display frame rate. In particular, the user can increase or decrease the number of original geometric primitives to be accessed and thus contributing to the extent of the localized fine detail volume being viewed.

- **"independently of the viewing direction":** This means that one can maintain high frame rates as described above, close to a model, far from a model, inside or outside a model, independentely of the viewing direction, or of depth complexity.

- **"independently of the size of the model in main memory":** This means that one can maintain similar high frame rates as described above with any size model that fits in main memory.

- **"specific graphics acceleration hardware":** Here we refer to the use of a high-end graphics acceleration hardware, such as for example at the time of writing an NVIDIA GeForce4 Ti which has capabilities beyond those of a standard graphics card such as an ATI Rage Mobility 128 graphics card with 8 Mbytes of SDRAM video memory.

1.3.1.2 Hypothesis #1.2

"A system can view such models without any manual pre-processing and with less memory than required for the original models without a perceptible difference."

- **"without any manual pre-processing":** This refers to existing visualization systems that require one to load smaller parts of a model and define texture boundaries that are compatible with the surface geometry of LoD blocks. We seek to create a system that does not require any such manual intervention.

- **"perceptible difference":** Lossless compression algorithms can offer no data loss at the cost of computation time. Here, 'without a perceptible difference' means that the rendering of the model that takes less memory does not differ from the rendering of the model's original version in a way that would affect a modelling task both visually or as to change the rendering speed in any way that would be noticeable.

1.3.2 Hypothesis Statement #2

"In the context of non-uniform reduction, there is a vertex classification system in which the same heuristics and constraints can be used with success with various popular uniform reduction methods and choices of error function. Such a system

can work with both regular, large scanned meshes and irregular, small CAD meshes and preserve borders and colour discontinuities"

- **"non-uniform reduction":** This means allowing uniform reduction in some areas whilst not reducing the resolution or level of detail in other areas, and managing the interface between the different areas.

- **"regular large scanned meshes":** This refers to models that are the result of dense, uniform sampling of a physical object by scanner systems.

- **"irregular small CAD meshes":** This refers to comparatively less dense models, perhaps generated by hand, and irregularly sampled, more densely in some areas than others.

1.4 Contributions

The main contributions of this thesis are divided into two groups, those arising from the first hypothesis and those from the second hypothesis. The main contribution of this thesis is the Interactive Octree Rendering engine that enables the inspection and editing of any model that fits in main memory without the memory overheads of LoDs and datastructures of current systems. Such overheads would force the models out-of-core with inherently slower access speeds and visualization delays.

We note that during an editing task, in order to precisely select a triangle from the screen with a device such as the mouse, a user requires a relatively close viewing distance to the model. In such viewing conditions most of the model is out of view and but a few thousand triangles suffice to fill the screen. Following this observation, our system first creates on the fly a clustered coarse mesh that we term the navigation skin. This skin provides an overview of the extent and shape of the model. Just as in the inspection of an object by means of a torch, the model's original geometry is then rendered outwards from the intersection point of the line of sight with the model by means of a depth buffer strategy over the navigation skin. The radius of the 'torch-beam' or magnitude of the triangle budget is controllable by the user at run-time. More powerful machines can support a wider beam radius or higher rendering budget. The system resorts to an array of pointers to volumes which we term the RenderArray whose distances to the intersection point of the line of sight are sorted every frame. The triangle budget is then spent on the triangles encountered in the first volumes of the RenderArray. The triangle budget provides a form of clipping and occlusion culling at no extra computation cost. Unlike visibility culling, where the amount of culled geometry can vary significantly creating large variations in frame rate, the number of volumes in the RenderArray being sorted is the same

every frame. The system times itself to find a level of the octree volumes that it can sort fast enough in order to provide an overall fast rendering frame time. Larger models or less powerful machines imply creating a shorter RenderArray. We found that our memory friendly process of creating an octree with in-place sorting of geometry creates volumes in which triangles were spatially coherent. This means that rendering triangles from sorted larger volumes still yields triangles that are close to the intersection point of the line of sight. In addition, our meshes were thus coherent and compared well with those produced by algorithms that prepare meshes for streaming [IL05].

We extended the Interactive Octree Rendering engine to tackle textured models. Unlike visualization systems that require manual intervention to align textures to geometry blocks, our system computes new textures for the volumes of the octree, allowing smaller volumes to access the textures of larger volumes and avoiding the problem of excess texture context switching between volumes of equal distance, or very nearly equal distance, from the line of sight. The number of volumes within the triangle budget are further sorted on texture id and distance before rendering.

The visualization engine Far Voxel [GM05] visualizes a variety of models such as CAD and scanned surfaces. Our method of creating and binding new volume textures could be used to extend Far Voxel to handle textured scanned surfaces too.

We tested our system with a variety of large models. We showed that it was possible to render a 28 million triangle model at the same speed as for a 10 million triangle model, in both cases from a variety of angles and, with a low end graphics card, at a speed at no time less than 15 frames per second. We present results with textured models and with isosurfaces models, and propose a solution for viewing CAD models.

Model attributes such as normals and colour can take up significant memory resources. Their compression allowed us to test the Interactive Octree Rendering engine with even larger models. We showed that an encoding scheme for unit normals using a subdivided icosahedron creates 2.5 times as many normals for the encoding than a subdivided octahedron for a given bit length and produces an encoding with lower error. This result enables one to use indices from a database of normals of two or four bytes length to look-up normals on the fly during rendering without using bitwise arithmetic that could take up computational resources from the CPU or GPU. A simple, colour look-up compression table is also presented.

The ability to interact with large models allows users manually to select features for non-uniform reduction. The contributions arising from the second main hypothesis are thus as follows.

We present a simple set of vertex classification heuristics that allow one to use different mesh reduction algorithms such as the quadric error on regular scanned models. We present a semi-automatic solution for the problem of condition number calibration in Chapter 4, together with a solution that deals with border simplification.

In order to be able to simplify a broad range of models, whatever their source, two geometric cleaning algorithms are presented. The first is an algorithm that consistently orients noisy, non-manifold surfaces. A simple algorithm for deleting duplicate vertices that would result in cracks in a LoD is also presented.

Finally, we show the benefits of non-uniform reduction in an animation application. For this application, an algorithm for creating automatically skeletons from body scans is presented.

In summary, the contributions of the first hypothesis of this thesis may be listed as follows:

1.4.1 *Octree Interaction Engine*

We created a new compact, memory friendly octree and hierarchical Renderarray. Its in-place creation does not use extra temporary memory that could cause a system to page with large models. This can be useful in several other applications. The in-place sorting also allows one to not have to store geometric primitives at leaf nodes with start and stop offsets used in each node instead. A side effect of the octree's in-place sorting of the triangle or vertex order is that we generate a mesh that is more coherent than the original which in turn is good for mesh streaming and compression [IL05].

1.4.2 *PNORMS*

We developed a new look-up table for triangle normals derived from different Platonic solids, with bounds on the maximum encoding error. We found that the icosahedron can yield a bounded maximum error below a tenth of a degree for demanding applications with 1:3 memory savings and that, when subdivided, it produces a more even distribution of vector normals than when other Platonic solids are used. This normal compression algorithm can introduce 1:6 memory savings without the need for decompression at run-time with negligible differences in rendering quality with 1.3 degree maximum error.

1.4.3 *A system that supports interactive viewing of large hierarchical CAD models, scanned models and texture data without manual intervention.*

The contributions arising from the second hypothesis and additional work may be listed as:

1.4.4 *Border quadric*

This is a quadric error based vertex placement strategy that does not over-constrain simplification at border regions.

1.4.5 *General vertex classification heuristics that allow for non-uniform geometry reduction*

This is a set of heuristics that work seamlessly with different popular error measures such as Garland's quadric error [GS97], Lindstrom's quadric approach [LT99], and linear placement. These heuristics can also be applied in the simplification of the boundaries of tetrahedral meshes [CM02].

1.4.6 *An automatic algorithm that calibrates the condition number of quadric LoD solvers.*

1.4.7 *A quantitative error evaluation that uses the same calibrated solver and condition tolerances between storing and accumulating quadrics following Garland's [GS97] quadric error approach and the memoryless quadrics approach of Lindstrom [LT99].*

1.4.8 *A cleaning algorithm to delete duplicate vertices.*

1.4.9 *An automatic algorithm that consistently orients surface normals.*

1.4.10 *A landmark based method that creates animation skeletons automatically.*

1.5 Publications Resulting from this Thesis

Six papers have been published in conferences and journals. The papers reflect different stages of progress of the work, with constant re-addressing of the question of the size of the 3D models used and scalability of the techniques developed. Renderings of simplified models using algorithms from the fourth and fifth chapters of this thesis for uniform and non-uniform geometry reduction respectively have been produced and featured in the following book:

M. Slater, A. Steed and Y. Chrysanthou, "Computer Graphics and Virtual Environments: From Realism to Real - Time" [SSC01].

We note that the recent publication of our colour and normal compression algorithm (PNORMS) has allowed us to collect and compile interaction results presented in this thesis with even larger models. We are confident that such results highlight the robustness and usefulness of the Octree Interaction Engine when viewing large models from main memory. A software development kit (SDK) with the PNORMS normal databases and encoding/decoding code has been available online since December 2006. In order to enable ply files describing large 3D models to be read on PC, UNIX or Macintosh systems some modifications to Greg Turk's ply reader were carried out. These tackle different line break conventions and are included in a second SDK that has also been made available online. A direct link from the

Stanford University 3D Scanning Repository page to the SDK has also been maintained. The publications in conferences and journals are as follows:

J. F. Oliveira and B. F. Buxton. Platonic derived Normals for error bound compression, In Proceedings of ACM VRST'06, November, 2006.

J. F. Oliveira, D. Zhang, B. Spanlang and B. F. Buxton, Animating Scanned Human Models, WSCG'2003, Journal of WSCG, Vol.11, No.2., pp 362-369, ISSN 1213-6972, 2003.

J. F. Oliveira and A. Steed, Determining orientation of Laser scanned surfaces, In proceedings of SIACG 2002, pp 281-288, 2002.

J. F. Oliveira and B. F. Buxton, Non-linear simplification of scanned models, In proceedeings of Numérisation3D-SCANNING2002, 2002.

J. F. Oliveira and B. F. Buxton, Light weight Virtual Humans, In proceedings of Eurographics-UK2001, pp 45-52, London-UK (runner-up to best conference paper prize), 2001.

J. F. Oliveira and B. F. Buxton, Creating light-weight virtual humans for Virtual Environments, Eurographics'99, pp 15-18, Milan-Italy(short article/paper), 1999.

In addition, the technical report below with preliminary results was made available on-line.

J. F. Oliveira and B. F. Buxton. An Efficient Octree for Interactive Large Model Visualization, University College London, Department of Computer Science, Tehnical report number RN/05/13, June 13, 2005.

1.6 Overview of Thesis

Figure 1.4 shows an overview of the system components developed in the work carried out in this thesis with arrows representing sequence dependencies. For example, for non-uniform reduction one requires the original 3D model to be cleaned for duplicate vertices whilst triangle normals have to be consistently oriented. The next step of inspection and interaction with the new Octree Interaction Engine allows for feature selection and a calibrated solver in the LoD system then produces a simplified model or sequence. It is possible interactively to inspect/select features without geometry reduction or geometry cleaning. Uniform reduction with a calibrated solver is possible without interaction or visualization.

The structure of the thesis manuscript is as follows. In Chapter 1, an introduction to the problems addressed and aims of this thesis is given. In Chapter 2 related work is presented and a detailed review of the quadric error is made. In Chapter 3 a new Octree Interaction Engine (contributions 1.4.1 and 1.4.3) is presented with a new compression algorithm (contribution 1.4.2) for triangle normals and colour. In Chapter 5 a new quadric for placement of vertices on borders is presented (contribution 1.4.4) together with heuristics for a general non-uniform geometry re-

duction (contribution 1.4.5) and an automatic algorithm for calibration of the numerical solver for optimal placement of vertices after an edge collapse (contribution 1.4.6). The chapter ends with a quantitative comparison between two popular quadric error based methods (contribution 1.4.7). The next Chapter 6, Section 6.1 describes the problems associated with inconsistent orientation of triangle normals and presents an automatic solution (Section 1.4.9, contribution 1.4.8). The chapter also presents a simple algorithm for cleaning duplicate vertices within a set Euclidean distance tolerance (contribution 6.2). In Chapter 7 we review the claims and applicability of the methods developed. Finally in Appendix A an application of non-uniform geometry is given in the context of an animation system which also includes a new automatic way of creating skeletons (contribution 1.4.10).

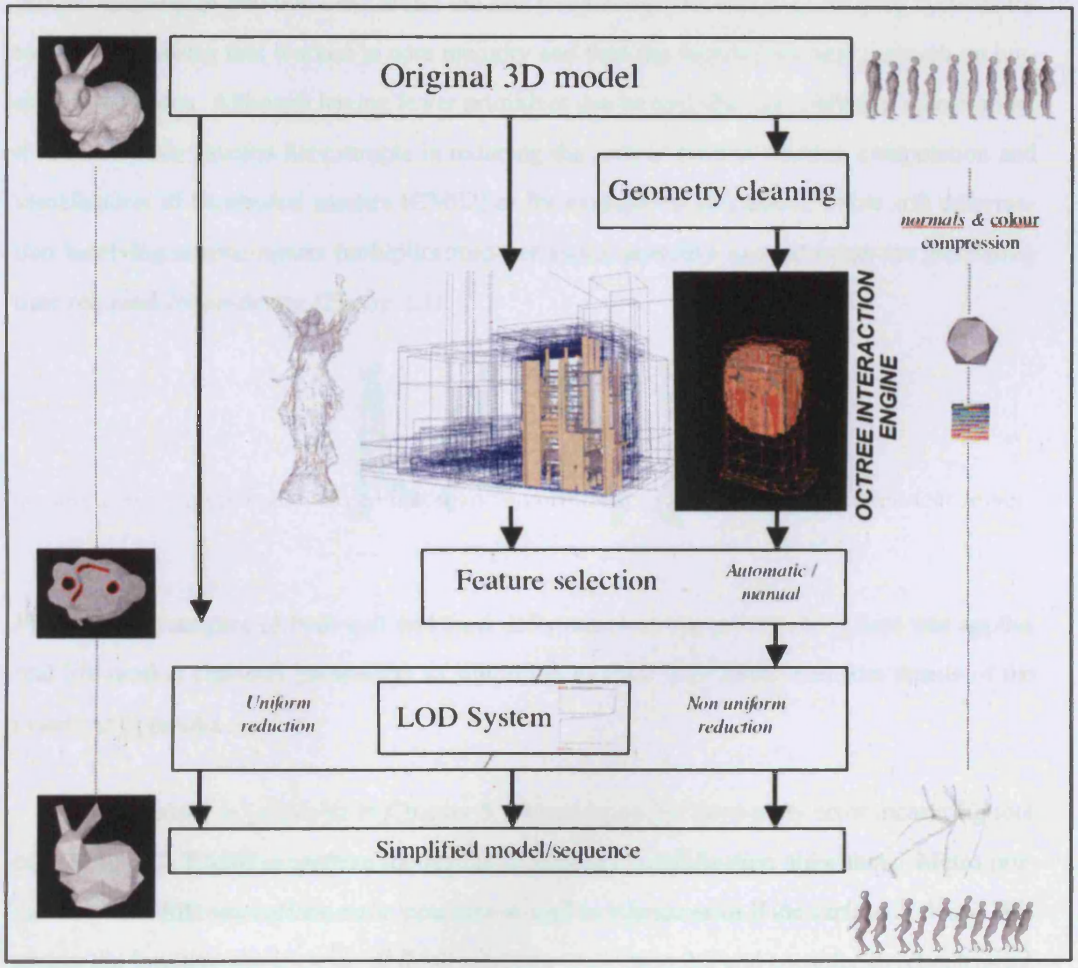


Figure 1.4: System overview

Chapter 2

Background & Related Work

Level of detail has been a very active area of research over the years. In this chapter we categorize the research into two main areas, the first research carried out on developing systems for reducing geometry that worked in core memory and then the second, the later research on out-of-core solutions. Although having fewer primitives can be desirable for rendering acceleration, it can also have benefits for example in reducing the time of both simulation computation and visualization of tetrahedral meshes [CM02] or for example in animation, where soft deformation involving several matrix multiplications per vertex primitive can outweigh the processing time required for rendering (Figure 2.1).

Figure 2.1: Examples of both soft and hard deformation in our animation system that applies real-life motion captured parameters to simplified meshes. The reader can find details of the system in appendix A.

In this thesis, in particular in Chapter 5, we use a popular third-party error measuring tool called Metro [CRS98] to analyze the results of different simplification algorithms. Metro provides several different surface error measures as well as volume error if the surface is closed. We review the intuitive significance of these measures in Section 2.2 and consider psychophysical aspects of the use of level of detail in computer graphics and virtual reality in Section 2.3.

These considerations are especially important to address at run-time and in view-dependent level of detail systems that have to decide automatically where to reduce the complexity of a scene. We review related work in Sections 2.4 and 2.5 and review in detail perhaps the most

popular LoD algorithm based on the quadric error at the end of the chapter (Section 2.6).

Later in Chapter 4 and Chapter 5 we show how the quadric error can be used to assess the geometric error when using different vertex placement strategies.

2.1 Basic level of detail system

Given an input model with several geometric primitives one can define a cost function based on surface distance, volume, or some other measure and associate a cost to each of the geometric primitives we wish to simplify be they: edges, faces, vertices, or voxels. As noted by Cohen et al. [CVM+96], there are two general approaches to reducing the complexity of a 3D model that we describe here:

- One approach is driven by the systematic reduction of geometric primitives in which the costs of each geometric primitive are inserted and sorted in a min-max heap and iteratively the smallest cost primitive at the top of the heap is chosen for deletion. We refer to this iterative loop as the *decimation loop*.

- The other approach is reduction driven by an error bound [CVM+96, CCMS96, ZM02]. These methods are fewer in number than methods based on the first approach and typically require some kind of auxiliary grid or spatial data structure to keep track of the simplification errors across the model. For example, Zelinka et al. [ZM02] use the generalized edge collapse of Garland [GS97] in conjuction with a voxel grid. Although error bounds are useful in the context of applications such as collision detection or physical simulations, the downside is that the number of triangles produced for a given error is unpredictable, often requiring some experimental trial and error to reach a compromise between triangle budget and error.

Most simplification methods follow the first approach. Perhaps the simplest error function is the edge length. Intuitively one does not wish to delete long edges that can contribute a lot visually to the appearance a model, so such edges typically end up at the bottom of the heap. After an element is selected for deletion, for example an edge being reduced to a vertex, the position of the remaining vertex can be calculated so as to be optimal with respect to the error function defined. Other methods of finding an optimal position might involve solving algebraic equations or a numerical solution such as singular value decomposition (SVD) for inverting a matrix and finding the zeros of a derivative. We shall see that numerical problems arise with certain geometric configurations. An automatic calibration method that addresses these issues is presented later in Section 4.3. After a local geometric simplification operation such as the

edge collapse [Hop96] on edge $\{Vt, Vs\}$ say (Figure 2.2, left: mesh M_{24}), one vertex Vt and two adjacent triangles $\{Vl, Vs, Vt\}$ and $\{Vt, Vs, Vr\}$ sharing that edge disappear (Figure 2.2, left: mesh M_{23}). The reverse operation is possible and is termed a *vertex split.* In this case a vertex and two triangles are added to the mesh. One problem that often arises when simplifing 3D models is the presence of duplicate vertices in memory (e.g. the same x, y, z vertex existing more than once in memory with triangles not sharing the same vertex index where they supposedly join). This situation is likely to happen, for example, in hiearchical CAD models in which separate objects are placed into contact via manipulation of their transformation matrices. In such cases, vertices are likely to be stored twice in each object definition or separator as specified in the VRML[1] 3D format and will have identical, or near identical, coordinates where they join. The consequence in the context of level of detail of having duplicate vertices $\{Vt, Vk\}$ and $\{Vs, Vh\}$ is that two edges exist ($\{Vt, Vs\}$, $\{Vk, Vh\}$) instead of just one $\{Vt, Vs\}$ and consequently, when one of the edges is collapsed, a crack is created (Figure 2.2, right: mesh M_{23}).

This problem can be hard to detect per se and even harder when object parts have accumulated numerical error in their individual transformation matrices making vertices rather less nearly identical and the difference variable or noisy. Another source of problems arising from potential duplicate vertices is the attribute management of a scene graph. A clever data structure for keeping the one x, y, z vertex with different material attributes such as colour or multiple vertex normals for *crease* management is given by Hoppe's *wedges [Hop98]* which utilises the concept of corners for preservation of *discontinuity curves* during simplification. This data structure assumes no duplicates or near-identical geometry. We present a new approach and algorithm for cleaning near-identical or duplicate vertices in Chapter 6. For reference purposes, we note that the object in Figure 6.35 is composed of many such joining cylinder-like objects that have duplicate vertices at the joins.

[1] Virtual Reality Modeling Language[WC07]

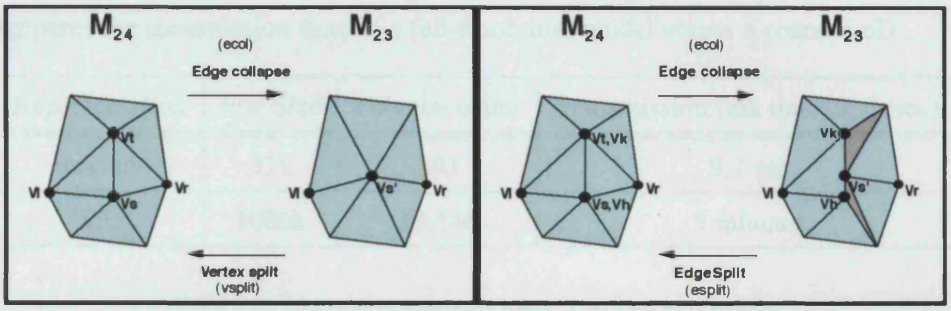


Figure 2.2: *Left:* the edge with unique vertices Vt and Vs collapses to vertex Vs'; *right:* in the presence of vertices Vk and Vh which are duplicates to Vt and Vs respectively, the *crack* in grey appears after the edge Vt-Vs collapses to Vs'. The vertices Vk and Vh belonging to an edge not connected to the edge Vt-Vs do not move to Vs'.

In order to carry out the geometric modifications to a mesh involved with an edge collapse one needs *local connectivity information* to retrieve affected neighbouring triangles and to re-label their indices from being relative to that of the deleted vertex to that of the other vertex that will have its position adjusted. Local connectivity is also needed to update the costs of affected edges. When we do so some edges might go further down the heap and some might go up in order to maintain the min-max heap condition [Knu73, Sed90]. In Section 6.1.2.3 we review in detail how an efficient local connectivity data structure can be built (presented by Garland in [Gar99]) and address several connectivity questions in Chapter 6, Section 6.1.2.3. After deleting a target number of triangles one can then save the simplified *static* model (Figure 2.3).

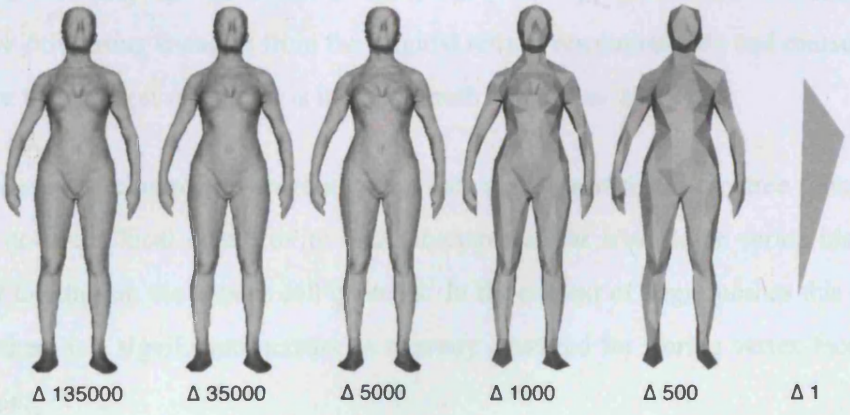


Figure 2.3: Static LoDs with decreasing number of triangles.

Applications such as on-line shops with 3D models representing their products can benefit from techniques such as geometry compression or from the use of coarse LoD models. Table 2.1

thus compares the transmission time of a full-resolution model versus a coarse LoD.

Representation	File Size	Polygon count	Transmission link time (3.3k/sec)
coarse	32k	393	9.7 sec
fine	1000k	13,546	5 minutes

Table 2.1: Internet download time: LoD vs full model.

If one wishes to do so, a list can be used to keep track of how the mesh was modified during an operation. For example, when vertices or triangles in a cluster are merged to an average position the triangle identifiers (IDs) that disappear can be stored in the record so as to facilitate the inverse operation if desired of adding them back. This may be carried out *dynamically* at run-time (Figure 2.4) using a level of detail controller according to some viewing parameter criterion such as distance, silhouette normal testing, or either a target surface error (Section 2.2) or screen space error (Section 2.3).

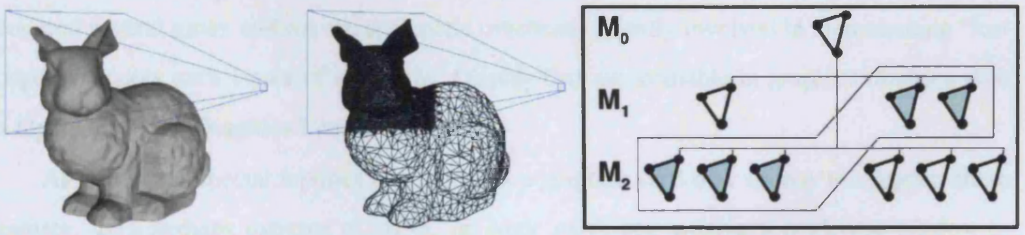


Figure 2.4: *Left:* Illustration of the use of dynamic level of detail using viewing parameters such as the view frustrum for *selective refinement* [Hop97]; *right:* Multi-resolution hierarchy with *active front* using triangles from the original refined resolutions M_2 and coarser resolution M_1, where the coarsest resolution is the base mesh denoted as M_0.

In the case of clustering, where a spatial data structure such as an octree [Sam90] is used one does not need local connectivity data structures as the triangle or vertex ids can be retrieved by looking-up the cluster cell contents. In the context of large meshes this can be very useful as there is a significant increase in memory overhead for storing vertex-face adjacency information.

Some out-of-core mesh simplification methods [Lin00] don't even store *shared connectivity* information. The meshes are said to be *unindexed meshes* or *raw*. Triangles are stored as triplets of xyz positions with the co-ordinates appearing several times in memory. By doing this, Lindstrom et al. [LS01b] report that the secondary memory requirements grow by a factor of two but claim a speed up of a factor of 15-20 in the simplification by not having to de-reference

vertices from triangles. In an *indexed mesh*, an xyz position should only occur once in memory and triangles are defined by triplets of vertex indices.

Making a *raw* mesh into an indexed mesh is essential for high quality geometry reduction and can be carried out by using, for example, the labelling strategy described by [CRMS03]. The reverse process, called *normalizing* a mesh, consists of converting an indexed mesh back to raw form, for example, suitable for clustering. This can be achieved via out-of-core sorting as described in [LS01b]. Many graphics acceleration cards actually have a data structure called a *vertex buffer* to which triangles are sent un-indexed in an attempt to minimize the slight time penalty of look-ups. *Triangle strips* are another data structure that improves rendering performance by not sending all the vertices or indices for each triangle. Continuous, connected strips of triangles are detected and positioned in order such that a new triangle in the strip only requires information about one vertex in order to be rendered as the other two vertices have already been provided by the previously rendered triangle. These strips have analogously also played an important role in mesh compression [TGHL98].

For completeness, we note that *display lists* are fragments of data that are ready to be rendered several times without the arithmetic overhead typically involved in incrementing "for-loops" to access each vertex of a triangle. Display lists are available in graphics libraries such as OpenGL (Open Graphics Library [WDS99]).

As with other special features or extensions a graphics card may or may not support these features. In a perhaps extreme example, on some machines resizing a rendering window to full screen will cause the graphics card to ignore such a request and the graphics library consequently resorts to using the central processing unit (CPU) instead to rasterize the larger images.

2.2 Surface error measures

We evaluate the geometric error of our approximation models by use of the public domain measuring tool, Metro [CRS98]. The tool provides measures of maximum error, mean and mean square error.

The distance from a point p to a surface S can be defined as the minimum Euclidean distance of the point to any point q on the surface S:

$$d_p = \min_{q \in S} \|p - q\| \tag{2.1}$$

Often one is interested in measuring the minimum distance for each of the vertices in the approximation mesh S_2 to the surface in the original mesh S_1 . Metro uses a simple spatial indexing data structure (e.g. uniform grid) to retrieve triangles from S_1 that are close to a query

point p on S_2. If the projection (p') of p onto a retrieved triangle plane is inside the respective triangle the distance $\| p' - p \|$is used as the minimum distance, otherwise if p' is outside a single edge plane of the triangle, the distance of p to that edge is used, finally if p' is outside two edge planes, the distance of p to a vertex common to the two edges used for the edge planes is calculated. In addition to considering all the vertices of the approximated surface for the minimum distance calculation, Metro also uses regularly sampled points inside triangles and along edges of S_2 . For details on their sampling strategy please refer to [Cig].

The largest of these recorded minimum distances over a surface can then be used as an upper bound on the minimum error. Note that this distance measure is not symmetric. If we were instead to measure the minimum distance of points of the original surface to the approximated surface one will obtain different results. This asymmetry is particularly important when sampling/projecting edge, face or vertex error as in [HDD+93, CRS98]. In constrast, the Hausdorff distance captures this error in a symmetrical manner by recording the largest minimum distance using both surfaces with respect to each other. The maximum error between the two surfaces can be denoted as:

$$E_{\max} = \max_i \|d_i\| \tag{2.2}$$

We found that this measure provided good feedback as to whether important structural information, such as a foot in our body models, disappeared or not on simplification. Metro actually calculates a signed distance using the normals of the original surface, so the mean error

$$E_m = \frac{1}{N} \sum_{i=1}^{N} d_i \tag{2.3}$$

can also be computed and we found that it provided good feedback on how well we were preserving the general curvature of the model. Similarly, the mean square error:

$$E_{msq} = \frac{1}{N} \sum_{i=1}^{N} d_i^2 \tag{2.4}$$

allows us to check how much on average our simplified surface has been offset from the original. An interesting illustration of the use of these measures is the observation (see Figure 2.5, left) of how the error in surface simplification can be reduced by not retaining the original vertices of a model. Obviously, if we are simplifying a purely planar surface it is best to move the vertices in the plane, but this is not so if we are approximating curved parts of a model. For example, in Figure 2.5 (left) if we keep the original end points the curve of radius 3 has a maximum error of 0.8 at the middle. However, if we allow both the vertices to shift outward

in the plane of the original curve, we can halve the maximum error and reduce significantly both the mean and mean square error. The quadric error approach automatically allows model vertices to move outwards or inwards as the level of detail is changed. However, it lacks the coordination of simplification operations needed to ensure that edges that might carry the same simplification error cost as edges elsewhere in the model are collapsed in such an order that symmetry is preserved. An artist solves this symmetry problem by manually breaking a model into several parts (in the case of the cylinder illustrated in Figure 2.5 (right) into 4 parts), simplifying automatically each part and then joining the identical simplified parts [Ste98]. In the case of preserving the bi-lateral symmetry of human body scans, some initial results with explicit symmetry constraints were presented by Oliveira et al. [OB99, OB01].

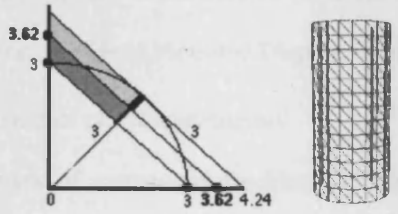


Figure 2.5: *Left:* Optimal maximum and mean error of shifted endpoints versus original endpoints of curve; *right:* Cylinder (for details see text).

2.3 Psychophysical considerations

Level of Detail can be exploited when well known visual phenomena occur with small loss of perceived image quality. Reddy [Red95, Red97] identifies these phenomena as:

- When an object is far away from the viewer, in particular when the area of the object projected into screen space is of the size of a pixel, it is pointless to apply matrix viewing transformation to a great number of polygons since the image fidelity will not increase. Figure 2.6, illustrates this principle. Comparison of the pixel difference of the original model's projected area and the simplified model's projected area is often called the *screen space error* and can include the pixel colour difference. Luebke et al. [LE97] use the *screen space error* at run-time to help the LoD controller decide where to reduce triangles in the model. They project the bounding sphere of a selected node representing a cluster of vertices in their scene graph as a measure of how much the vertices can move during the clustering of that node. The projected diameter of the bounding sphere represents the worst case of the screen space error associated with the node when, for example, the two vertices that are furthest apart in the cluster are collapsed into one. If the sphere is far

away from the viewer this projection leads to very little pixel error whilst if it is close then significant pixel errors can occur and the cluster needs to be unfolded into smaller nodes with smaller projected bounding areas to meet a set screen space error. This fast run-time technique has also been used in view dependent out-of-core techniques such as GoLD [BGB+05].

- When an object is moving fast, high frequency, detail acuity is also diminished.
- When the perceived image of an object lies in the periphery of the viewer's retina, visual acuity is also not as sharp as in the area centred around the fovea.

There are two types of systems for tracking the gaze direction of the viewer in order to allow for acuity degradation in the periphery of the field of view, those:

- based on the orientation of a Head Mounted Display Helmet, and
- directly from measurements of eye movements.

Extensive coverage and review of current eye tracking techniques is given by Martin Reddy in [Red97]. Reddy also stresses the need to quantify visual changes in rendered images so that psychophysical information can be applied to make level of detail both more realistic and more efficient.

Predicting a viewer's position based on velocity has been an integral part of modern out-of-core visualization systems that constantly have a pre-fetch thread that uses predictions of what the user is going to see next to load ahead of display time the required geometry from disk. As noted above, visual acuity diminishes when the viewer is perceiving moving objects without directly fixating on them. Reddy suggests that this degradation can be quantified by:

$$Temporal\ Frequency = Spatial\ Frequency * Angular\ Velocity$$

An example of a dynamic LoD system that uses spatial frequency to control the level of detail being rendered is the system presented by [LHNW00] in which the user's gaze is tracked and used in the rendering parameters. In this system, the silhouette of an object is rapidly detected and preserved by querying view cones that contain and represent several normals in a given area of the model. If the angle between the viewer's direction and the axis of the view cone is 90° then those areas are not simplified in order to increase the perceived quality of the simplification at the silhouette boundary as noted by [XV96, LE97].

Viewing parameters play an increasingly important role in prefetching out of core geometry for frames to be rendered in the future. Borgeat et al. [BGB+05] pre-fetch geometry immediately outside the viewing frustrum by deliberately considering a view cone that is larger than the

one used for view frustrum culling. Psychophysical experiments have been carried out by using user search tasks to investigate the effect of resolution reduction in the periphery of the field of view [WWHW96]. These experiments revealed that the time users took to perform the task was not affected by degrading in the peripheral vision area the resolution of 2D images of the object being sought to half its original resolution. Care has to be taken however in controlling the level of detail switching and ultimately the frame-rate variation. The problem of fluctuating frame-rate can be very disturbing to an immersed user when for example he or she is trying to grab an object in the virtual world or to place an object in a specific position [WSNR96].

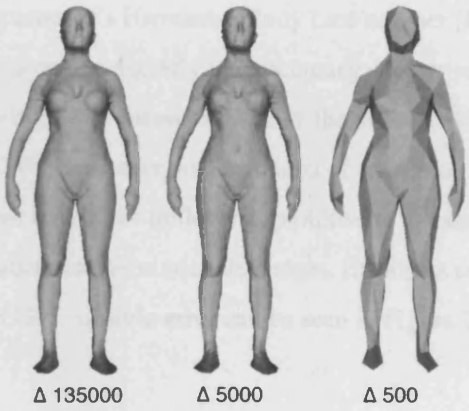
In particular, this study reveals that in systems that deliver rendering speeds of the order of 20 fps (50 ms) changes of frame rate of up to 40% of the average frame rate do not affect user's performance of a task in a virtual environment. Another imporant pschophysical aspect is the perceived appearance of a model, in particular where appearance can have scalar attributes such as colour, polygon normals, material attributes (such as *bump maps*[2], the way the material reflects different wavelengths of light, etc.), or textures. Parts of a model that have a sudden change in such attributes are often called *discontinuity curves* [Hop96]. The task of preserving discontinuity curves becomes harder as polygon count decreases. One can argue however as to whether a particular discontinuity should be maintained when the polygon complexity falls. For example, for a complicated model like the Cessna [Hop96], at a low polygon count of below hundred polygons (say), should one of the hundred polygons be dedicated to represent the discontinuity of a small window in the original model when it only contributes to a single pixel in the screen when viewed at lower resolution far away? Hoppe allows simplification across discontinuity curves at that level by associating a cost to the operation that is less than the cost of breaking the geometry of say the join of the wing to the fuselage. Tuning these parameters is not a straight forward task however. A similar argument can be raised concerning small, connected components of models, such as the antenna of a car, as observed by Cohen [Coh99]. Perhaps the topology should be simplified as in [ESV98, ESV99] and the antenna removed.

A method that preserves discontinuity curves in a less explicit way is that of Cohen et al.'s appearance texture map [COM98] that encodes normal and colour information at full resolution. Using texture-mapping hardware, simple geometry is rendered by later sampling the texture map light or colour information. Klein has yet a different approach to preserve appearance attributes such as the depth of a specular highlight in a smooth surface. Klein [KSS98] keeps track of curvature and keeps polygons where their normals could potentially be needed to ensure

[2]bump maps is a technique that tries to emulate the appearance of complex geometry when using simple polygons, it uses 2D images with pixel intensities that mimick the lighting of normals computed with real geometry.

the fidelity of a smooth highlight but reduces the mesh more drastically elsewhere.

Some computer games aim for a target polygon count in the worst case scenario and try to maintain frame rate. Attempts such Funkhouser's [FS93] LoD load balancing algorithm (reviewed in detail later in Section 2.4.2) are more general and allow for other run-time processes to be included.



Left: Original 135,000 triangle model, *middle:* 5,000 triangle simplified model; *right:* 500 triangle simplified model. At a close distance differences in detail are perceptible, but here flat shading has been deliberately employed to highlight that the models are indeed different. Gouraud shading would make the 5,000 triangle simplified model even more imperceptibly different from the original model.

Left: Original 135,000 triangle model, *right:* 5,000 triangle simplified model. Both models are viewed from the same further distance from the viewer.

Left: Original 135,000 triangle model, *right:* 500 triangle simplified model. Both models are viewed from an even greater viewing distance than in either of the above.

Figure 2.6: Original model and LoD rendering comparison.

2.4 In-core solutions

We note that many level of detail algorithms that were designed to create alternative discrete resolutions of a model with fewer primitives can usually be extended to have the history or sequence of geometric modifications used also in a dynamic solution. In this section we review in-core solutions in the settings in which they were presented.

2.4.1 Static solutions

A static solution can be regarded as a one-off simplification model for a target triangle budget or target error for a specific, given application. For example, the original resolution whole body scans produced by the department's Hammatsu Body Line scanner [Hor98, Hor95] can be used to calculate the volume of a person digitally to an accuracy of approximately less than 5% error as determined by comparison with measurements of the volume from water displacement by the human subject [DDBT99]. However, in the context of animation such size meshes were, on commodity machines at the turn of the millenium, prohitive to simulate or to render and need to be reduced to a much smaller resolution triangle budget. Examples of different static resolution models using Garland's [GS97] quadric error can be seen in Figure 2.6.

Vertex removal

Schroeder et al. [SZL92] have reduced models using a three step method in which they:

1. Characterize the local vertex geometry and topology into 5 different types - Simple, Complex, Boundary, Interior Edge, and Corner. Complex vertices are essentially non-manifold vertices, where multiple surfaces touch. These vertices are not removed. Oliveira et al. [OS02] present a variant of the Complex vertex type, the *Complex Boundary* vertex which is important to detect in the context of the triangulation of the border of holes.

2. Delete vertices using a decimation criterion dependent on the type of the geometry. Each vertex that is deleted creates a hole where the triangles connected to the deleted vertex used to be.

3. Re-triangulate the resulting hole using a splitting plane and keeping the original positions of remaining vertices.

This method preserves sharp edges but can distinguish and eliminate spikes from noise in, for example, polygonal mesh models obtained from CT data. Each vertex that may potentially be removed has an error associated with it. This error is calculated from the distance to an average plane defined by the area weighted summation of the normals of the original triangles joining at

a vertex divided by the local total triangle area. In the case of border or crease vertices, the error is calculated from the distance to an edge formed by the two endpoints connecting to the vertex. This local error criterion was later extended by Ciampalini et al. [CCMS96] who use both local and global versions of this kind of error to drive the simplification. They use the maximum of two such error measures to determine the maximum global error in a patch and the maximum of all these maximum errors is the global error of the mesh. The first error measure uses vertices within an approximation triangle as samples to calculate the distance to the average plane of the original surface. The maximum of these distances is recorded and added to any existing triangle error. We note in passing that Guéziec [Gue96] similarly dynamically computes new error volumes incrementally using already computed errors. The second error measure traces the approximation triangle that is nearest to the vertex being removed and records the maximum vertex-face distance of removed vertices and their nearest approximation triangle. Hoppe et al. [HDD+93] also use extensive sampling to track errors. In this case, essentially the sum of three energy functions is minimized whilst performing edge collapse operations:

$$E(M) = Edist(M) + Erep(M) + Espring(M)$$

where $Edist$ is the squared distance of samples after a simplification operation, $Erep$ is the number of primitives used, and $Espring$ is an energy function that ensures that the minimization converges to a minimum (for details see [HDD+93]). Edges joining triangle pairs could have one of three geometric operations applied, *edge collapse*, *edge split* that would insert a vertex into the edge and thereby create further edges, and *edge swap* that would replace an edge with an edge connecting a pair of vertices opposite to the edge in order to improve the triangle aspect ratio. This algorithm is known in the literature ([CCMS96], [Gar99], [Coh99]) to produce the best approximations in terms of geometric error but it is computationally expensive. Later, a faster version was created [Hop96] that only performed the edge collapse geometric operation and two energy functions were added, one being $Edisc$ and the other a new energy $Escalar$ which together help preserve *discontinuity curves* between different material attributes or creases. Hoppe's progressive meshes were ideal for view independent refinement. A sequence of edge collapses would generate simpler models independently of the viewer's gaze and these simplified models could then be used to speed-up computation in a target application. In this work, the concept of *geomorphs* was also presented in which vertices being collapsed could be alpha blended to reduce a popping effect that occurs when switching between different static LoDs. Finally, early view-dependent experiments on terrain data were presented by adjusting a progressive mesh according to view parameters. The proximity of triangles with

respect to the view frustrum was taken into consideration for refinement operations and the size of projected triangles on the screen was also used to guide refinements or simplifications. A more complete solution to dynamic view dependent refinement [Hop97] by Hoppe is reviewed in the in-core dynamic category.

Turk [Tur92] also removes vertices, re-distributing fewer vertices over a mesh and shifting them by *repulsion* over the surface so that new polygons are formed from the new vertices and some of the old vertices. This method generates good aspect ratio triangles and can be regarded as both a mesh simplification method and a surface re-meshing method in which new meshes are created to improve the regularity (explicit re-mesh), aspect ratio, or mesh fairness[3] [Kob97].

Another notable contribution presented in [SZL92] is the recursive split-plane re-triangulation algorithm that closes the surface after a vertex is removed. The border vertices of a resulting hole form a loop. A splitting plane is created such that it is orthogonal to the average plane in the affected area defined from the normals, centroids and areas of the triangles around a vertex. The splitting plane is set to pass through two non-neighbouring vertices in the loop, thereby generating two loops of boundary vertices. The split plane is then passed again through each loop until there are just three vertices in the loop. There are several ways to pass the splitting plane through a loop so a test is carried out in order to choose the splitting plane that would yield the best triangle aspect ratio. In this test, the distances from the border vertices of a loop to the split line are summed (Figure 2.7; bottom right) and divided by the length of the split line. If the resulting number is large then the splitting plane yields triangles with good aspect ratio.

Other re-triangulation methods are possible. A brief description of each one mentioned in Figure 2.7 is given here. Delaunay triangulation yields the triangles with best aspect ratio as a triangle formed in Delaunay triangulation is such that there are no other vertex points within the circles that circumcircle each of its edge (dashed). If there is a point within that circle, then the old edges(dashed) are deleted and new edges connecting to a sample point p are linked. More details about Delaunay triangulation are given in [Lis94]. The benefits resulting from Delaunay triangulation in height field[4] terrain data can be seen in [COL96]. Coarse representations of the height field in this example are Delaunay triangulations, new finer meshes are also Delaunay

[3] *Mesh fairness* is an important measure for measuring the quality of surfaces, for example, in the context of the automative CAD industry as it will dictate the numerical quality of a simulation [GSS99]. Surfaces not only have physically to connect (C^0continuity) but their higher order derivatives also have to be continuous and smooth C^1 and C^2. Continuity and smoothness can be evaluated both at the geometry level and at a parameterisation level [GSS99].

[4] A height field is often regarded as 2.5D (dimension) data in which two coordinates define a position on a global plane or patch, plus an offset/height.

triangulations, and are enclosed in the coarser level triangulations. In the second example, triangulation of the Museum view commences with any border vertex and links it to all the other boundary vertices in the line of sight - a technique which can yield long thin triangles. In the third method, a local greedy-algorithm explores the proximity of boundary vertices within the list - a technique which can also yield long thin triangles. In the fourth, a list traversal scheme then tries to link vertices in the list as long as they don't generate intersecting edges. When three edges are connected a new vertex is considered for linking a new triangle - yet another technique which can yield long thin triangles. To offset these effects the length ratios criterion (fifth example) is used as a 'rule of thumb' when creating new triangles. A test is carried out on the ratios of the lengths of the sides of a triangle and if any of these ratios exceed a threshold, indicating that the triangle would be long and thin, the triangle is rejected and an attempt is made to form a triangle with the other remaining vertices around the hole. If there are none left, then the long, thin triangle has to be kept.

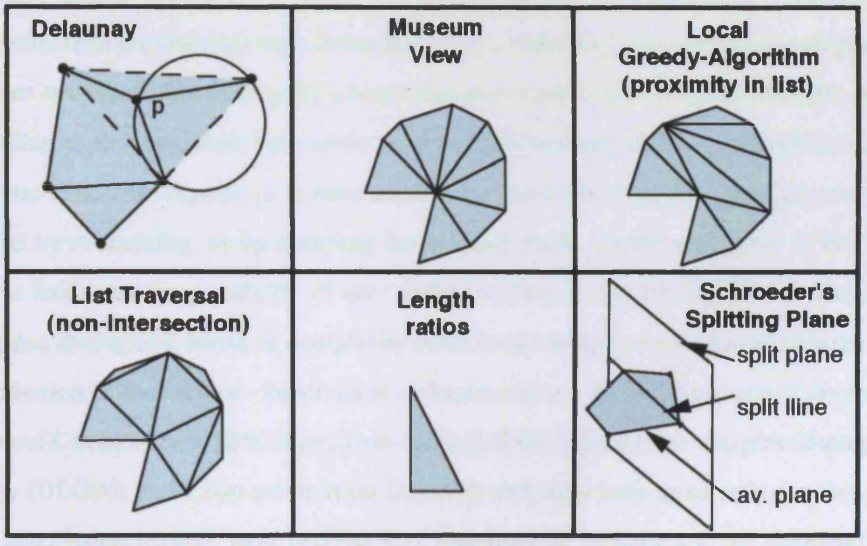


Figure 2.7: Hole re-triangulation schemes.

Multiresolution analysis (MRA)

This approach first creates a coarse base mesh by deleting vertices from the original mesh by means of an averaging process. Then sucessive differences between the original and coarse meshes are encoded as detailed information in *wavelets*. Eck et al. [EDD+95, Lou94] have developed a multiresolution analysis level of detail system that works with arbitrary meshes. Their method generates good, low complexity representations of curved objects but unfortunately considerable detail needs to be present if the way it approximates almost flat surfaces

is to work well. This problem has been alleviated by Lee et al. [LSS+98] by using an angle tolerance measure to detect creases. The boundaries of surface patches are aligned with creases which might indicate the start of a flat region. Gross et al. [GGS95] provide an adaptive meshing algorithm based on a wavelet approach and a multilevel quadtree placed over a regular grid. Cracks that might otherwise result from triangulation of regions of different adjacent resolutions are prevented by coding 16 possible look-up triangle configurations.

These methods require very little memory relative that is, to approaches that store all level of detail representations in main memory. If more detail is needed in a particular area, wavelets are simply added. One disadvantage of multiresolution methods in general is that they might require more triangles than the original object if a surface is to be smooth and might require the input mesh to be re-meshed if it is to be semi-regular in terms of topology or how the vertices connect to form triangles. Recently Igor et al. [GVSS00] use a parameterisation over a semi-regular mesh that is created from a mesh of arbitrary topology. Their normal surface representation allows vertices in a model to be spatially defined by means of a single offset float value rather than the standard three floats for x, y, z. Often in the presence of multiple objects in a scene one would like to simplify a base mesh even further. Igor et al.[GSS99] use sucessive edge collapses to create their base mesh. A strong pre-requisite of these approaches is that the model has to be *semi-regular* or to have *subdivision connectivity* [GVSS00]. This can often be achieved by re-meshing, or up-sampling the original mesh. Another property of this method is that it facilitates the possibility of user surface editing at any level of detail. Changes are propagated through all levels of complexity since local changes can be traced through the parameterisation of the surface. Subdivision surfaces/schemes derived from quad polygons such as those of Catmull-Clark [BS02] and Doo-Sabin [DS98] derived from triangles (Butterfly subdivision [DLG90] and Loop subdivision [Loo87]) and from both quad polygon and triangle (Loop subdivision [SL02]) have become very important for modellers in the computer animation industry (DeRose et al. [DKT98], and Stam [Sta00]). It is not clear how such subdivision engines scale with large models. A view-dependent subdivision approach could utilise the view frustrum to limit the number of subdivision steps but whether it can provide a user dependent, granular localized control at run-time is not apparent.

Clustering

Rossignac et al. [RB93] place a *regular grid* on an object to find vertices in close proximity to each other. Vertices within a grid volume are then merged into a cluster, representative vertex. Schaufler et al. [SS95] use a BSP tree to query vertices that are within a dissimilarity distance to create a hierarchical vertex cluster tree. This tree is then used to create discrete LoDs using

as their first discretisation step from the root 1/8th of an object's bounding box that contains all the scene, etc. down to finer levels. The results of distance dissimilarity queries are stored in the *binary space partition tree* (BSP tree) and the BSP tree is then used to link triangles to representative cluster vertices at the set discretisation step. Schaufler et al. also present a viewer-based approach that allows switching between discrete LoDs in which the level of detail representation to be used is determined by projecting the bounding box diagonal into screen space. At run-time their system determines the potentially visible parts of the tree by testing the bounding boxes with the planes of the view frustrum. Schmalstieg [Sch97] use octree quantization for vertex clustering to create discrete VRML representations of an object. Octree quantization is more sensitive in its spatial adaptation to the geometry of an object than uniform grids are. Schmalstieg reports that octree quantization produces better quality approximations than uniform quantization of regular grids. He also reports that octree quantization is faster than the creation of BSP trees. For LoD switching, Schmalstieg proposes to use the distance from the viewer to the object and some range values that represent the maximum deviation or distance that a vertex travels during clustering. He uses a fraction of the extent of the cube at a given level of the octree to which he adds the radius of the bounding sphere of the cluster to compute range values that can be used at run-time. Finally Schmalstieg uses the octree to clean duplicate vertices in a model. Care is needed when designing an algorithm to delete duplicate vertices as the order in which a vertex is deleted can prevent other vertices from being deleted. In Chapter 6, Section 6.2, we present a two-pass vertex deletion algorithm that ensures that all vertices within a tolerance from another vertex are deleted.

DeHaemer et al. [DZ91] presented two approaches (Polygon Growth, Adaptive Subdivision) for reducing the complexity of quadrilateral polygons in regular CT scanned grids. As in Schroeder [SZL92] they also address noise inherent from the scanning phase of data acquisition.

Their polygon growth approach commences with a seed polygon and tries to group neighbouring polygons to form one, flat polygon that does not exceed an error criterion. With the adaptive subdivision approach one starts with a large trial polygon and then recursively subdivides it into two or four smaller polygons at the point in the trial polygon with the highest degree of error when approximating the surface. If the polygons formerly introduced still have areas that exceed an error tolerance that area is again subdivided (Figure 2.8).

Hinker et al. [HH93] and Kalvin et al. [KT96] have also presented methods for simplifying flat or homogeneous areas of objects. Groups of homogeneous shapes are created and simplification is carried out within each group. We note in closing that implicitly the boundaries of these areas dictate the shape of the object at low levels of detail.

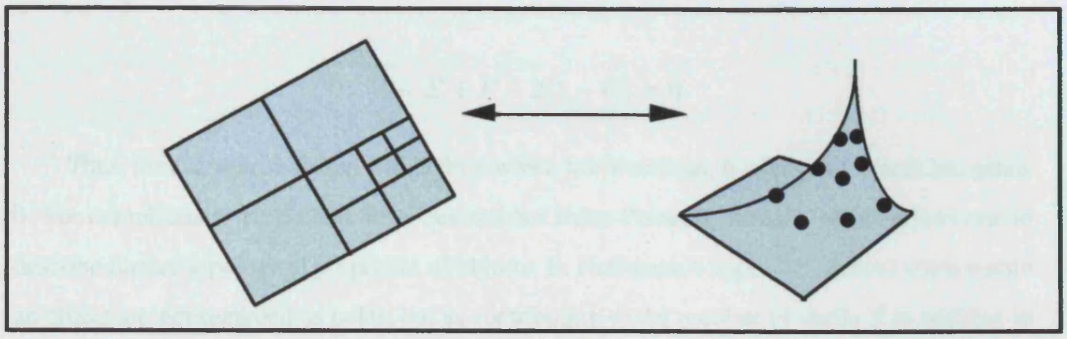


Figure 2.8: DeHaemer & Zyda[DZ91] adaptive subdivision. New quadrilateral polygons are added recursively where the error between the original surface and the trial polygon exceeds a threshold.

Topology simplification

The word topology is often used to denote how vertices are connected locally to form triangles. Such topology is described, for example by terms like the following: a regular quad topology as encountered in terrain data; a semi-regular topology as in subdivision surfaces where the edge valence at a vertex, i.e. the number of edges connecting to a vertex is almost the same throughout the object (a vertex with different edge valence is said to be an *extraordinary vertex*); an irregular topology referring to objects that do not have a consistent valence throughout, and arbitrary topology used to denote that an algorithm makes no assumption on the local topology configuration of an input mesh or is prepared to change that topology. However topology also means the structure of an object and is related to the geometric *genus*. In *constructive solid geometry* (CSG), objects are represented by Boolean operations performed on basic solid primitives comprised, for example, of the sphere, cylinder, cone, torus, parallelepiped, and triangular prism [Hof89]. How they are tesselated is not represented. Boundary representations define how surfaces are oriented and tesselated and follow Euler's formula:

$$V - E + F - 2 = 0$$

A simple closed object such as a sphere has genus 0 [Hof89], a torus or a coffee mug which has one handle has a genus 1, and so forth. Sometimes it is useful to compare the genus of objects. Objects are *homeomorphic* if they are topologically equivelant or have the same genus. The genus of a surface is the number of handles that the surface has [Hof89]. An object with two holes is topologically equivalent to a sphere with two handles, hence it has genus 2. Each hole in an object increases the genus of an object by one. In general, the genus of an object can be calculated by means of the Euler-Poincaré formula:

$$V - E + F - 2(1 - G) = 0$$

Thus, for example, a closed tetrahedron which has 4 vertices, 6 edges and 4 faces has genus 0. For completeness Hoffmann describes another Euler-Poincaré formula which allows one to describe further topological properties of objects. In Hoffmann's approach internal voids within an object are not regarded as holes, but as contributing to the number of shells S in addition to the surface shell, whilst loops or bounded geometry such as single vertices, lines, and each face, count towards the number of loops L of geometry. These concepts can be used to calculate the sum genus of an object, which as it is a generalisation of the genus is still represented by G, according to:

$$V - E + F - (L - F) - 2(S - G) = 0$$

For a cube, for example which has 8 vertices, 12 edges, 6 faces, 6 loops, and 1 shell, the sum genus is zero, same as the genus obtained from the Euler-Poincaré formula. It is important to note that an object, such as a cube, with some kind of depression or cavity within a face will have an extra loop to account for the vertices of the cavity where it interfaces with the face in which the cavity is embedded.

It is challenging to determine the genus of some objects. However, as long as one does not tear or glue parts of an object, one is allowed to twist, enlarge, push, and flatten an object [She06]. A coffee mug with one handle can have the rim squashed to the disk at its base and its handle twisted alongside that base, the hole of the squashed handle counts as genus 1 and hence the genus of the coffee mug is one. Some objects obey the Euler-Pointcaré formula but are not topologically valid, non-manifold configurations, such as that illustrated in Figure 6.12 which count as just one shell.

He et al. [HHK+95] place a voxel grid over a mesh and apply marching cubes filtering to voxel buffers. Popovic et al. [PH97] reduce the genus of an object by clustering progressively simplicial complexes that contain local neighbour geometry. He et al. [HHVW96] voxelize a mesh and use a marching cube filtering process to reconstruct meshes with simpler topology.

El-Sana et al. [ESV98] use α-prisms to detect salient features, holes and cavities to reduce the genus and topological complexity of an object. Cubes of width 2α are centered at each vertex of a triangle in the mesh, the three axis-aligned cubes form an α-prism when the cubes are convolved along a path that encompasses the three cubes. Candidate features, such as borders of a hole and creases are detected to form sets of tesselation lines, the α-prisms on

the tesselation lines are intersected with the neighbouring mesh to determine whether a hole or inaccessible part exists. Once inaccessible parts are found, a new tesselation is computed to fill the holes and reduce the genus. El-Sana et al. show how surface simplification algorithms such as [CVM+96] can be used with their feature detection and explicit topology reduction to increase an overall reduction of the number of triangles in a mesh for faster visualization. They point out that for mesh reduction techniques to be used in finite element methods with CAD models, the genus of such objects cannot change. In other words, structural elements such as holes, tunnels and cavities in a model have to be preserved during simplification to ensure that the topology or genus of the object is preserved and thus the object can still be used in the application.

Still in the context of mesh reduction in finite element applications, Chopra et al. [CM02] preserve the shell of their tetrahedral volumes and use a tetedrahedral fusion operator to obtain larger reduction steps than would be obtained by edge collapse performed on individual tetrahedra. Finally we point out that Garland [Gar99] introduces the generalized edge collapse by detecting *virtual edges* that, within a proximity criterion, connect previously unconnected geometry. By collapsing virtual edges, the genus of an object can also be reduced.

Simplification Envelopes[CVM+96]

This method is best described perhaps as a generalization of offset surfaces. Basically two envelopes are created and wrapped around the object. One has a positive offset and the other a negative offset that for a convex object would usually place it within the object.

They present two criteria for applying the simplification based on the two envelopes and the original input mesh:

1. Given an error value, create a mesh with a minimal number of vertices that does not exceed the error measure at any point.

2. Given a number of vertices, create a mesh by distributing those vertices across either envelope, choosing the vertex combination that yields least error.

This method creates good quality approximations and is later used in [COM98]. In this work, Cohen et al. separate an object into three different channels: geometry, colour/texture map, and bump maps. The bump map is formed from a 2D array containing the original polygonal normals. This array is then sampled along with the colour map array onto a simpler geometric representation. The result is designed to have (some of) the lighting effects of a full-resolution model present in coarser models by using pixel-hardware support. Although good approximations are achieved with simplification envelopes, the method requires the object's surfaces to be

well oriented. In Chapter 6.1 we examine the problem of orientation consistency and present an automatic solution.

Quadric error methods

Ronfard et al. [RR96] track the maximum perpendicular distance of a vertex to a set of planes. Initially each vertex has a zero error distance to the planes of its incident triangles. When two vertices are merged by edge collapse, a linear system of equations is used to determine its optimal position and the surviving vertex inherits the planes of the other deleted vertex. Tracking these planes can, for larger models, be both prohibitively expensive both computationally and in memory requirements. Garland et al. [GS97] keep track of the squared distance to planes using a 4×4 homogeneous matrix. The squared distance of a vertex to a plane is coded by use of a 4×4 matrix called the 'fundamental quadric'. For any given vertex typically several fundamental quadrics can be calculated, one for each triangle that shares the vertex. Conveniently fundamental quadrics can be added and stored as one quadric matrix. This matrix when pre and post multiplied by a homogenous position vector gives the maximum squared distance of that vertex to a set of planes. When vertices are merged after an edge collapse, the quadric matrices of the edge's endpoints are simply added. Lindstrom et al. [LT99] further show that these quadric matrices do not need to be stored at all to produce good approximations. The quadrics are temporarily computed based on the current geometry and deleted. Garland extended the quadric error matrix to higher dimensions to optimise colour and textures [GH98].

Hoppe [Hop99] introduced a new quadric that relates to colour deviation in R^3. Specifically, another cost is added to each fundamental quadric that represents the deviation between a point's colour and the interpolated colour of a triangle's three vertex colours at the point's projected position. Hoppe also allowed discontinuity curves amongst attributes to be preserved at vertex level within the quadric error framework using a wedge data structure that stores multiple attribute data in a vertex. Recently the quadric error was extended to arbitrary dimensions [GZ05]. Since we derive a method of simplifying border edges based on the quadric error, we review the quadric error in detail in Section 2.6.

2.4.2 Dynamic solutions

The majority of dynamic LoD viewers test either the geometry or the bounding boxes of parts of a model against the four planes of the view frustrum. Clark [Cla76] used view frustrum clipping and switched between discrete LoD representations of objects in order to speed up rendering. View frustrum culling or clipping can greatly reduce the number of geometric primitives to be rendered, as can occlusion culling [TS91, BSGM02]. However, the potentially visible set

that may need to be rendered can still be quite large in many instances. Teller et al. [TS91] precompute cell visibility information in very occluded scenes, as for example typical of indoor walkthroughs, creating portals that can then be queried at run-time thereby eliminating a lot of occluded geometry.

However one problem with static heuristics for visibility determination [COCSD03] and LoD selection is that the number of polygons rendered during each frame depends on the size and complexity of the objects that are visible making the frame-rate arbitrarily slow or non-uniform [FS93]. In order to obtain a constant frame rate Funkhouser et al. [FS93] optimize a cost-benefit function applied to the portion of objects that have passed the view-frustrum culling test. Each object (O) has two sets of tuples associated with it: benefit tuples for each of the object's LoD (L) with a particular rendering style (R), and the corresponding rendering costs. A perfect cost heuristic may depend on the model and features of the graphics workstation, and different rendering styles such as Gouraud shading, Phong shading, or texture mapping have different costs and benefits. A perfect benefit heuristic would consider human perception, occlusion, and colour [FS93]. Benefits for each Object/LoD/Rendering style can be adjusted at every frame by semantic importance (walls in a scene might be more important than other objects, for example), by focus (objects near the middle of the screen can be attributed a higher benefit than those in the periphery of the field of view), by motion blur (objects moving fast can have lower benefits for higher detail), and by hysteresis where level of detail switching can be discouraged if switching becomes too erratic over a sequence. Each of these benefit factors range from 0 to 1 and are calculated as follows:

$$\begin{aligned} Benefit(O,\ L,\ R) = \ & Size(O) * Accuracy(O,\ L,\ R) * Importance(O) * ... \\ & ... * Focus(O) * Motion(O) * Hysteresis(O,\ L,\ R) \end{aligned}$$

The idea is then to maximize the sum of benefits, subject to the sum of costs being smaller than a threshold required for a target frame rate. Finding the optimal tuples is NP-complete, so Funkhouser et al. use a greedy algorithm instead that computes the value of each tuple:

$$Value(O,\ L,\ R) = Benefit(O,\ L,\ R)/Cost(O,\ L,\ R)$$

For each frame, a new working set of visible tuples is computed. Objects that existed in the working set of the previous frame have the same LoD and rendering style as in the previous frame whilst newly visible objects have their LoD and rendering style set to lowest cost options. The accuracy of the highest value tuples are incremented first, and then the lowest current value tuples have their value decremented, until a target cost is met. Funkhouser et al. compare

four different dynamic view-frustrum culling algorithms using the same walkthrough test path through a complex indoor auditorium:

1. *No detail elision* i.e. no LoD, objects are rendered at their highest detail, with simple view frustrum culling.

2. *Static* with simple view frustrum culling and static discrete LoD switching according to the size of objects as projected on the viewing screen. Objects whose average polygon contributes to more than 1024 pixels in the screen are rendered at their highest detail.

3. *Feedback* with simple view frustrum culling and static discrete LoD switching where the projected size tolerance is changed taking into account the time difference between the previous frame time and the target frame time.

4. *Optimisation* view frustrum culling in which an appropriate LoD and rendering style is predicted and chosen for the visible objects to match a target frame rate.

Frame-rate graphs for 1, simple view-frustrum culling with no detail switching, show frame rates that are generally long and non-uniform since the frame rate depends directly on the complexity of the objects that are visible. For 2, switching between static LoDs with view frustrum culling increases frame rate considerably but preserves a similar frame-rate curve pattern. The 4th method produces a more uniform frame rate than the 3rd method, partly because the 3rd method is a reactive feedback system that cannot handle sudden changes in complexity which the 4th can. By means of a lazy increment of cost and benefit of a working visible set, the 4th method has the ability to over-ride the importance of detail of up-close objects to meet a coherence/hysteresis criterion. Funkhouser et al. also report that varying rendering style is less disturbing to a viewer than varying frame-rate. In this work they use two processors, one for visibility and LoD-switching and the other for rendering.

The MAVERIK system [HCK+99] allows immediate mode switching of different rendering styles, such as to a wireframe mode to limit the load. Hubbold et al. parameterize objects to control the number of triangles/LoD to be rendered at run-time. The MAVERIK system supports bounding volumes, hierarchical bounding volumes, gridcells, and hierarchical gridcells, and the VR kernel accepts different call-backs for adding functionality. The system also provides a measure of system stress [HDG96]. Other VR functionalities essential for interaction, such as collision detection, were used in an emergency drill planning scenario in VR [HK00].

So far, most of the dynamic methods described have performed static LoD switching between a limited set of discrete, simplified representations of a model. A linear sequence or

history of edge collapses can be useful for view-independent construction of a discrete, simplified mesh to a target triangle budget. However, for managing detail in the context of navigating through data in a view dependent manner a linear sequence of collapses would not adequately preserve regions of interest. For example, for a particular area to be simplified or refined, a long history of edge splits or collapses would have to be performed. Xia et al. [XV96] build a merge tree of vertices where different levels of a binary tree are created from the original leaf vertices upwards to a single coarse point. Each level of the tree represents the object entirely and, using object-space edge collapses, vertices at either end of short length edges are merged first. At run-time an object can have parts rendered using different levels of the tree by using a datastructure termed an *active front* that keep track of the various resolution levels used to render the model. To avoid cracks between different LoDs in adjacent parts, the tree is built with some *dependencies*. For example, vertex pairs are merged as long as the regions they affect does not overlap with another area in which edges have already been modified. During the construction of the tree, two Euclidean distances are stored in each node of the tree, an up-switch (coarsening) distance and a down-switch (refinement) distance that relate respectively to the surface error between the modified and unmodified meshes. At run-time the screen projection of this distance is used to determine whether a part of the tree can be simplified or not. Initially all original triangles are deemed displayable then, with each adjustment to the tree, triangles are removed or added for display. Puppo [Pup98] encodes the dependencies of different simplified parts of a model using a *directed acylic graph* (DAG). The collection of these dependencies is called a Multi-Triangulation. Combining different parts of the graph generates different triangulations. At the root of the graph lies a coarse mesh. Guéziec [GTLH98] presents a compact way for storing dependencies by keeping track of the direction of edge collapses.

Luebke et al. [LE97] also build a vertex tree hiearchy but use an octree to allow merging of disconnected vertices. This is particularly useful for CAD models that are comprised of a disjoint sets of objects. To create a more balanced octree that allows more uniform run-time access, Luebke et al. use a *tight octree* that adjusts/tightens the extent of a cell so as to enclose all the vertices within it, before sub-dividing the cell again. They allow the creation of high quality surface-based simplification trees/forests of a number of single objects or parts of object(s) to be topologically merged into a single tree. The cone of normals approximated during the creation of the vertex tree can limit the merging of objects.

At run-time three criteria are used to select vertices to triangulate and render from the tree. Luebke et al. compare the view direction against the normals of cones that represent several normals within a local area rapidly to determine wether a part is in the silhouette of an object

and needs more resolution. They use a screen space error based on the projection of the extent of the bounding volume of a node in the tree. Finally, we note that Luebke et al. allow the user to control the triangle budget of renderable triangles by minimizing the maximum screen space error of bounding volumes for that budget. *We note that whilst the user can control the available renderable triangle budget, there is no guarantee of rendering original resolution triangles at any part or sub part of a model. In the context of marking features for non-uniform geometry reduction a user needs to visualize and select for preservation original resolution triangles. Furthermore, an increase in triangle budget can result in the triangle budget being spent across the model rather than on the region at which the user is directly looking.*

Later more perceptual measures such as the calculation of the spatial frequency of a moving object were added by Luebke et al. in [LHNW00].

As with other vertex-tree methods the process of selection of the appropriate depth in the tree of viewable vertices is separated from the actual rendering process. These two processes are *asynchronous* and can be parallelized over multiple processors.

Hoppe [Hop97] also builds a vertex tree but instead of using the edge length criterion of Xia et al. [XV96] for merging vertices, Hoppe uses as input an arbitrary progressive mesh (edge collapse sequence) which minimizes a better simplification metric based on sampling and minimizing distances to the original mesh. Hoppe filters region *dependencies* at each operation to build each level of the tree. Some concern has been raised [ESV99] about the run-time cost for checking for crack-free dependencies and of mesh fold over prevention tests at run-time. Multi-triangulations [FMP98] is a data structure that stores dependencies between LoDs enabling fast querying of different LoDs in adjacent areas at run-time. El-Sana et al. [ESV99] store dependencies implicitly by generating new vertex ids that have a higher number than all the existing vertices before a vertex merge. At run-time these ids can limit or allow simplification by checking the parent ids of the boundary vertices of the affected region against the vertex id of the parent of the edge. If the id of the parent of the edge is larger, then the simplification can take place. Conversely a vertex can be split if its vertex id is higher than the vertex-ids of all its neighbors. The method presented by El-Sana et al. [ESV99] is one of a few that allow for object aggregation by controlling both topology and surface simplification. A subset of Delaunay edges of a 3D Voronoi diagram are included as generalized virtual-edge collapse (topology simplification) candidates for the merge tree.

Erikson et al. [EM00] present a method for creating hiearchical levels of detail (HLoDs). Standard topology reduction techniques are used to create LoDs of objects, then approximations of groups of objects (HLoDs) rather than of just single objects in a scene graph are created. The

LoDs of children of the node are used to form the HLoDs of the node. At run-time the tree thus does not need to be further traversed. Erikson et al. argue that whilst adaptive refinement of vertex trees at run-time provide useful capabilities, it also imposes a significant overhead at run-time in terms of memory and CPU usage. Hence HLoDs switching allows for drastic switches in detail which are necessary when visualizing large CAD objects such as the Double Eagle Tanker [YSGM04] consisting[5] of 126,630 objects, and 82 million triangles.

Software provided by Riegl's 3D Laser scanners [Rie07] render a scattered triangle subset of a model when the user is interacting with it and once the user stops moving, the complete set is rendered.

2.5 Out-of-core solutions

Larger models derived from scanned data have presented a challenge to conventional rendering systems as the data created exceeds by a large amount the main memory available [LPC+00]. In this section we review initial work that addressed these meshes and briefly review the state of the art of out-of-core rendering systems.

2.5.1 Static solutions

Lindstrom [Lin00] sorts vertices in secondary memory and defines a grid size that is suitable for the RAM available. One pass finds the bounding box of the mesh and then triangles that have their vertices completely within a grid cell are removed. Quadrics for each vertex are stored in a separate file and are added together to form one quadric per grid cell. Vertices within a cell are removed and a new single position representing the whole cell is computed using the quadric of that cell. The triangles that survive are the ones that have vertices across different cells. This method uses the *cluster vertex representative* idea [RB93] and uses the homogeneous property of the quadric to add matrices without having to track connectivity.

Sorting external meshes plays an important role in making secondary memory access more localized and coherent which is essential in the context of view-dependent out of core access. Isenburg [IL05] presents several techniques for re-labelling vertex and face indices so as to make a model *streamable* over a network and locally coherent. Making out-of-core meshes more coherent also reduces storage space through out-of-core compression [IG03]. Cignoni et al. [CRMS03] create meshes that have vertex indices that are more coherent by re-labelling the indices of the vertices of triangles according to the lexicographical name of the octree cell in which the vertices lie. Oliveira et al. [OB05] create meshes that have coherent triangles owing

[5]Their system achieved 1-8 fps with a selected view-path, rendering the model in the core memory of a 16 GBytes SGI Reality Monster.

to the in-place sorting of triangle positions within the triangle array during the construction of their octree.

Lindstrom et al. [LS01b] moved intermediate level of detail data structures out-of-core as well allowing for the target clustered model size to be independent of the RAM available. A further contribution from this work was a technique for border preservation without using connectivity information. The idea follows from the fact that two adjacent triangles sharing one edge have two counter-clockwise half edges on each side. A half-edge quadric built with a plane perpendicular to the incident triangle (as in [GS97]) is added to every half-edge of the mesh. Consequently border edges and sharp edges are preserved without using connectivity information which would be costly to create in memory. For manifold edges of non-degenerate triangles the half-edge constraints will cancel in opposing half-edge quadrics. A method that temporarily computes connectivity information of out-of-core meshes so as to enable tasks such as editing is presented by Cignoni et al. [CRMS03]. One inherent problem of grid clustering is that the resulting meshes will have the size of their features determined by the initial grid spacing [LS01b]. Shaffer et al. [SG01] use quadrics and an adaptive binary space partition tree [Sam90] (BSP) to distribute cells non-uniformly over an out-of-core mesh. A different approach by Fei et al. [FCGW02] uses multiple passes to create their simplified meshes. They calculate *principal curvature* from quadrics [Gar99] and re-introduce vertices in areas with high curvature and along creases after each simplification pass on the mesh.

Cignoni et al. [CRMS03] created an octree-based external memory mesh (OEMM) data structure that allowed for editing and mesh simplification of models that do not fit in main memory. An overview of the out of core model is provided using wireframe octree bounding boxes of nodes higher up in the tree. Clicking on these boxes allows a user to select the part of the mesh to load in core, in a block fashion. Read&Write flags are stored in vertices and triangles to control what geometry can be modified based on what has been loaded into main memory. Incident vertices or triangles that are not within a cell have their corresponding ids stored in the vertices and triangles to which they link. A global indexing scheme is presented that re-labels vertices using the ids of their octree nodes. A second pass is made to label vertex indices of triangles that have vertices that are not within the octree node. The mesh simplification algorithm of Cignoni et al. creates several disjoint local priority queues, one for each subtree node using the quadric error [GS97] metric. Subtree nodes are visisted in a lexicographical order and the subtrees are simplified separately.

2.5.2 Dynamic solutions

Aliaga et al. [ACW+98] present a walkthrough system that is viewpoint-cell driven. For each new cell entered, static LoDs, precomputed visibilty information, and texture-depth renderings are pre-fetched from secondary memory. They use pre-computed cell visibility [TS91] and place several cull boxes on the model for fast occlusion culling outside these boxes. They use view-frustrum culling for scene graph nodes inside the cull box. Static LoDs for the visible geometry are chosen according to distance and further occlusion culling is performed using a two-pass method (occlusion selection followed by culling) called hiearchical occlusion maps (HOMS) [ZMHKEH97]. The visible geometry inside the box is then rendered and finally texture-depth-images of far away geometry outside the cull box are mapped to the faces of the occlusion box.

QSPLAT [RL00] uses a hierarchy of bounding sphere volumes for fast visibility culling and LoD selection based on splatting different size pixel points rather than polygons. Their method is well suited for rendering scanned models comprised of 100 million to 1 billion samples. They point out that whilst point based rendering of uniform scanned models is faster than rendering a polygonal representation, polygon rendering of models with large and flat areas produces better visual results. The same is true if zooming up-close between samples. At run-time, high-resolution data is paged in from disk and a reactive/feedback LoD system such as those described in Section 2.4.2 is used to meet a target frame rate set by the user.

Corrêa et al. present iWalk [CKS02a] for visualizing large out-of-core CAD models. In this system, a scene is spatially partitioned into an octree, visible octnodes are paged-in to memory or re-used from cache and, instead of pre-computing cell/region visibilty information [TS91], a smaller point visibility set is computed at run-time. Two visibility algorithms are used. One is an approximate mode which uses a prioritised layered projection of potential visible octnodes (plp). This process is bounded by a triangle budget and is fast. The other is a conservative mode [KS01] which is more accurate and uses a Z-buffer to fill the potential holes from plp. This method is slower because it needs to page-in the geometry of all the deemed visible nodes. In [CKS02b] the authors parallelize iWalk to use a cluster of PCs. The 2D screen is divided into tiles and, at each frame, geometry is assigned to each tile and rendered by a separate CPU. We note that these visibility algorithms are ideal for paths inside highly occluded models. For paths outside the model the authors suggest the use of LoD techniques [KS01]. *It would be desirable to combine the speed characteristics of the approximate mode with the accuracy of the conservative mode in a seamless manner.*

El-Sana et al. [ESC00] store a vertex merge tree in secondary memory for view dependent

refinement of out of core meshes in which subtrees can be paged into memory and meshed. Similarly DeCoro et al. [DR02] presented a solution (XFastMesh) in which a base mesh is always present in main memory. Subtrees of the vertex tree are represented in a compact, block file format and are paged-in to triangulate finer areas along with some coarse triangles from the base mesh. Whereas previously models of 8 million triangles could not be visualized at any interactive rate, these solutions achieved rendering speeds of 4-5 frames per second with some constraints on the viewer. Reported visualization paths were comprised of views that were always up-close to these models. View frustrum culling at these distances eliminates the majority of the model and a steady path allowed for coherence in the caching of pre-fetched data.

Synchronous out-of-core pre-fetching of view dependent data from disk imposed a direct cost on the rendering throughput. Recently ([CGG+04], [YSGM04], [BGB+05], [GM05]) decouple rendering from out of core fetching. This asynchronous strategy allows the graphics card to render at full speed always what is stored in the card's video memory cache and prevents the graphics pipeline from stalling. A coarse model can be always readily available in the graphics card video memory and out of core updates are either inserted in a priority queue and phased-in, or are deliberately limited in scope and thus allow the rendering to be GPU bound.

One of the key aspects that makes the *clustered hierarchy of progressive meshes* (CHPM, [YSGM04]) and hierarchical surface patches of TetraPuzzles [CGG+04] and [BGB+05] successful is that the original scene/surface is represented as a limited set of smaller manageable clusters/sub-surfaces. These sub-parts make complete switches between discrete LoDs that represent the sub-surfaces at different levels, rather than using more fine grain switches at edge level in a mesh. Borgeat et al. [BGB+05] present a method for hardware *geomorphing* between these different sub-surface LoDs. Some manual work is necessary to segment textured meshes but thereafter their system efficiently pages in out-of-core data.

Far Voxels [GM05] uses a binary space partition tree to partition a model into smaller manageable parts and build a regular cubical volume grid. Each voxel of the grid contains alternative view-dependent appearance representations of the scene from different angles. Sample rays are cast from each volume onto the geometry using the BSP tree to enable the creation of parametric shaders. These parametric shaders are used to splat a vertex representing the voxel when the voxel size is smaller than the tolerance set by the user. A priority queue sorts voxels from front to back and occlusion culling is used to accelerate the rendering. Whilst this method can be used as a powerful visualization tool, enabling the inspection of out-of-core scanned models and CAD, it does not support editing of a model, and does not support texture scanned

models.

We note that these out-of-core methods have been specialized for mainly one or two types of 3D models. TetraPuzzle's hierarchical tetrahedral volumes and Far Voxel can partition and visualize a scene that consists of scanned surfaces or CAD, but not textured models. In addition Far Voxel's rendering approach can be used to visualize iso-surface extracted models whose topology would be difficult to simplify with conventional simplification methods. CHPM [YSGM04] addresses directly CAD models whilst GoLD [BGB+05] addresses textured surfaces, but not CAD.

2.6 The Quadric error reviewed

As mentioned in Section 2.4.1, the quadric error metric has proved to both be robust and to match the high quality approximations of models produced by slower methods based on sampling [Hop99]. We review this metric here, as several aspects from the quadric error are addressed in Chapter 4.

Given the zero inner (dot) product of two perpendicular vectors:

$$\vec{N} * \vec{a} = 0 \tag{2.5}$$

a plane with normal $\vec{N}$, and a point (x_0, y_0, z_0) on the plane can be expressed by means of the following equation:

$$N_x * (x - x_0) + N_y * (y - y_0) + N_z * (z - z_0) = 0 \tag{2.6}$$

This equation can be written in a more familiar form as:

$$ax + by + cz + d = 0 \tag{2.7}$$

If we normalize $\vec{N}$ by dividing it by its length, we satisfy Garland's [GS97] condition:

$$N_x^2 + N_y^2 + N_z^2 = 1 \tag{2.8}$$

The distance of a point to the plane is then just a projection of the point's position vector onto the normal. Since, in a matrix notation with vectors represented as 3-component column vectors

$$N^T * v + d = 0 \tag{2.9}$$

we note that:

$$d = -N_x * (x_0) - N_y * (y_0) - N_z * (z_0) \tag{2.10}$$

If we re-write Equation 2.9 again in matrix-column vector form as:

$$D = p^T * v$$

with

$$D = \begin{bmatrix} a & b & c & d \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.11}$$

then the distance squared of a point to the plane is:

$$\begin{aligned} D^2 &= \left(p^T v\right) * \left(p^T v\right) \\ &= \left(v^T p\right) * \left(p^T v\right) \\ &= v^T * \left(pp^T\right) * v \\ D^2 &= v^T * K_p * v \end{aligned}$$

which defines K_p . In component form:

$$D^2 = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.12}$$

Initially a vertex has zero distance to the set of planes that meet at that vertex. The *fundamental quadric* $\mathbf{K}_p$ from which the squared distance to a single plane may be computed can then be added to those of other fundamental quadrics $\mathbf{K}_{p'}$, say, to form $\mathbf{Q}$. The squared distance to a set of planes can thus be calculated by just post and pre multiplying a single matrix, and can be written as:

$$\begin{aligned} D^2 &= v^T \left(\sum_{p \in planes\, of\, v} \mathbf{K}_p\right) v \quad , \textit{or simply} \\ D^2 &= v^T \mathbf{Q} v \end{aligned} \tag{2.13}$$

which in turn defines $\mathbf{Q}$.

The general algorithm computes $\mathbf{Q}$ once at start-up for every vertex in a model. It then considers all edges in the model and computes a cost for each edge by temporarily adding to the

quadric $\mathbf{Q}$ for the first endpoint the quadric $\mathbf{Q}$ of the second endpoint of the edge. The summed quadric describes an ellipsoid in 3D, as can be seen by writing Equation 2.12 in Cartesian form. In doing so it is convenient to represent the elements of $\mathbf{Q}$ formally as in Equation 2.12.

From Equation 2.13 one thus obtains:

$$D^2 = a^2x^2 + 2abyx + 2aczx + 2adx + b^2y^2 + 2cbyz + 2dby + c^2z^2 + 2cdz + d^2 \quad (2.14)$$

We are interested in the point (x, y, z) in Equation 2.13 that corresponds to the minimum value of the sum of squared distances to the set of planes around both vertices. This point can be found by taking the partial derivatives of Equation 2.14 and setting them to zero:

$$\begin{aligned} \frac{\partial D^2}{\partial x} &\equiv 2a^2x + 2aby + 2acz + 2ad = 0 \\ \frac{\partial D^2}{\partial y} &\equiv 2abx + 2b^2y + 2bcz + 2bd = 0 \\ \frac{\partial D^2}{\partial z} &\equiv 2acx + 2bcy + 2c^2z + 2cd = 0 \end{aligned} \quad (2.15)$$

This can be expressed as $Ax = b$ in a homogeneous, 4-vector notation as:

$$\begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad (2.16)$$

The optimal point can then be found by inverting A , for example by using Cramer's [Str88] rule, or by Gaussian elimination with pivoting:

$$Ax = b$$

$$A^{-1}Ax = A^{-1}b$$

$$x = A^{-1}b \quad (2.17)$$

Numerical problems arise in the presence of parallel planes (or very nearly parallel planes) in the quadric as a single, unique optimal position does not then exist and A is singular (or very nearly so). A test on the magnitude of the determinant uses a fixed tolerance (e.g. 1×10^{-12}) that is usually encoded into the system. If the determinant falls below this value, then finding the optimal position along a line or computing an average position can be attempted. Alternatively, a numerical solver such as *singular value decomposition* can provide a condition number that diagnoses the health of the quadric; i.e. how poorly conditioned computation of its inverse

will be. We found that this number is not only application dependent but also sensitive to the measurement scale of a model (Section 4.2). In Chapter 5 we propose an automatic solution.

After each edge collapse has had the potential cost of its associated optimal position stored, these costs are sorted in a min-max heap. When the smallest error edge is collapsed and removed from the heap, connectivity information is used to determine the triangles that share the edge and will be deleted. Other edges that will be relabelled to use the remaining vertex will have their costs updated in the heap.

The quadrics can be area weighted for tesselation invariance with the area of each triangle divided by three and used to pre-multiply a fundamental quadric [Gar99]. Border constraints can be added to stop borders receding. This is accomplished by inserting a plane perpendicular to the triangles containing the border edge and, instead of the area, the length of the border edge times 1000 can be used to multiply the fundamental quadric to be constructed. These fundamental quadrics formed from the perpendicular planes are added to the initial quadrics of the border vertices. The factor of 1000 ensures that such quadrics are, for all but very short edges, stored a long way down the min-max heap and the corresponding edge features thereby preserved until the later stages of the LoD simplification. In Chapter 5 we propose vertex classification heuristics as an alternative way of dealing with borders and features.

Additionally vertices deemed important can be discouraged from moving by using point quadrics. Instead of the distance to a plane, the direct distance to a vertex's original position can be used:

$$D^2 = \begin{bmatrix} x & y & z & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & -v_z \\ -v_x & -v_y & -v_z & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.18}$$

where D is the distance function of a generic vertex (x, y, z) in R^3 to a given vertex (v_x, v_y, v_z). This point constraint can also be added to the initial quadrics.

Chapter 3

Octree Interaction Engine

Non-uniform geometry reduction methods assume that an operator can efficiently interact with a triangle model held within main memory and mark features manually for non-uniform reduction. Before we can explore a generic framework for non-uniform geometry reduction (Chapter 5), we need to address the issue of interactivity with medium to large models. Rendering acceleration techniques such as LoD can confound the selection of triangles from the original model. The memory overhead required to maintain connectivity and hierarchy data structures necessary for LoD switching force several of the example models, described later in this chapter, out-of-core. Further, delays inherent during paging geometry from secondary memory are undersirable for extensive editing tasks and other applications running on more limited hardware, for example a PDA (portable digital assistant) or mobile phone.

Our main aim is to have compact data structures that allow immediate access to the local model geometry in a manner that is rapid enough for models of any size residing in main memory. To achieve this, we developed a new interactive visualisation system which we term the Octree Interaction Engine (OIE). This chapter presentes the development, refinement and testing of the OIE.

Our system is essentially comprised of a memory friendly octree which creates a volume based hierarchy and a RenderArray data structure which is essentially an array of pointers to volumes of the volume hierarchy. The Octree Interaction Engine chooses automatically an appropriate level of the volume hierarchy to operate efficiently with the given computational resources available and the complexity of the loaded 3D model. The system sorts the RenderArray's pointers to volumes according to the distances of the volume's corners and their centre to the intersection point of the line of sight and the model. The triangle rendering budget is prioritised to be rendered from the first volume pointers of the RenderArray. Unlike gaze directed systems [LHNW00, WWHW96] that use computational resources to find the vector product of the view direction and hierarchical mesh surface normals to refine or coarsen the whole mesh,

our prioritisation and triangle budget limit provides ***implicit*** visibility [COCSD03] and occlusion culling.

This is advantageous with respect to view dependent acceleration techniques where the rendered load may be assigned a constant budget for every frame independently of view-scene complexity. The visualization of scanned models with our system is augmented with the rendering of a coarse *navigation skin* mesh that provides an overview of the extent of the model. The navigation skin is created by clustering [Lin00] the original mesh on the fly just after loading the model and the creation of the octree.

A depth buffer strategy ensures that the original geometry is always rendered over the navigation skin or other augmented visualization modes, such as the rendering of bounding boxes, so that it does not interfere with the selection of triangles from the original model.

Cignoni et al. [CRMS03] used wire frame rendering of hollow octree bounding boxes that could be selected by the user to load in-core a part of a mesh. We similarly allow the user to preview and adjust both the level of the scene's wire frame bounding boxes and the level of octnode wire frame boxes. One difference is that Cignoni et al.'s [CRMS03] external memory manager (EMM) loads the complete mesh within a selected block, whereas we allow partial meshes within a fixed triangle budget around the intersection point of the line of sight and the model to be loaded instead. Thus we allow immediate inspection of areas across different blocks.

The base system of the Octree Interaction Engine is described in Section 3.1. The base system in addition to rendering geometry from the model can optionally render the octnodes of the tree at different depths chosen by the user. This render mode is particularly useful when editing multiple surface objects such as the white and grey matter of the brain model (Figure 3.5) where simple clustering tends to create disturbing self-intersections of the navigation skin. The base system of the Octree Interaction Engine presented in Section 3.1 was also documented in [OB05] and was later further developed to handle large textured models.

Extensions to the base system are necessary to handle a variety of other different types of model, such as textured models, scanned surfaces or CAD models. Each of these model types has intrinsic modelling differences or challenges that need to be considered in order to apply the scalable rendering paradigm of our base system. Texture models follow an individual surface rendering approach which necessitates changes to the rendering cycle and initialization in order to stop and control the rendering budget on multiple indexed volumes of different textures (Section 3.3). Scanned surfaces can benefit from a coarse clustered mesh to provide an overview of the extent of the model (Section 3.2). The visualization of CAD objects composed of several

sub-objects uses additional scalable augmented visualization techniques and is discussed in Section 3.4. Synchronous rendering performance and our hypothesis of maintaining high frame rates at any viewing angle is tested for different types of model in Sections 3.1.5, 3.2, 3.3 and 3.4 .

In order to test the performance of the Octree Interaction Engine with models that do not fit in main memory, a new real-time capable algorithm for the compression of normal and colour attributes that do not require reconstruction is presented. Our compression algorithm which we term "Platonic Solid derived normals for error bound compression" (PNORMS) was published in [OB06] and is presented in Section 3.5.

3.1 Octree Interaction Engine - base system

In this section the core elements of the Octree Interaction Engine are described. The differences to conventional LoD rendering systems (Figure 1.2) are visible in Figure 3.1. These differences stem from our aim of keeping models in core memory with immediate access from viewpoint directions PA to PB to the fine resolution geometry M4. We do not create or store LoDs M2 and M3. An array of pointers to octree volumes which we call the RenderArray is sorted every frame. Sorting is carried out according to the minimum distance of a volume's corners and centre to the intersection point of the line of sight and the model. A triangle budget set by the user is then rendered from the first volume pointers in the RenderArray. Sorting and calculating these distances for a large array of pointers to leaf volumes of a deep octree can be computationally demanding. The initialization step measures the time it takes for the available computational resources to compute the distances and sort a RenderArray. The system repeatedly builds and destroys RenderArrays until finding an adequate pointer-volume size. Different temporary RenderArrays are visible in Figure 3.1 between M4 and M1 along D. In the case of scanned models a coarse clustered navigation skin M1 is kept in main memory.

In Section 3.1.1 we provide a brief background to octrees. In Section 3.1.2 we describe how we build our memory friendly octree. Once the octree is built the RenderArray can be built (Section 3.1.3). The subsequent rendering process is described in Section 3.1.5. An initialization step (Section 3.1.4) is used to adapt the Octree Interaction Engine to the existing computational resources.

Our in-place sorting of geometry during the octree construction has the somewhat surprising effect of making the mesh more coherent and applicable to mesh streaming (Section 3.1.7). A detailed description of the editing task requirements of the brain model and presentation of the editing tools developed to help triangle selection provides an illustrative example application

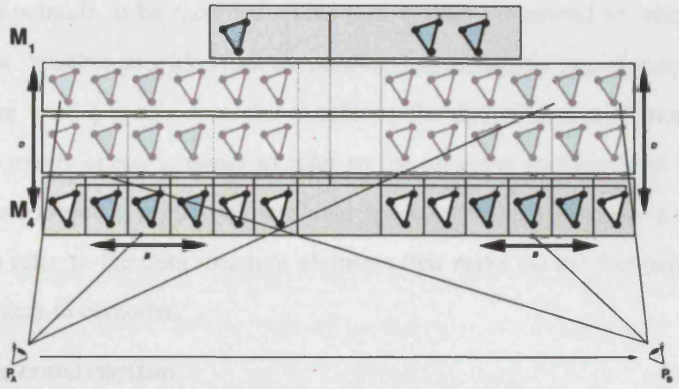


Figure 3.1: Octree Interaction Engine architecture. The original model M4 is accessible without delays from viewpoints PA and PB. Both the triangle budget B and depth D of the RenderArray of pointers to volumes from which the triangles are rendered are adjustable at run-time. For scanned models a small simplified mesh M1 can provide a permanent overview.

(Section 3.1.6).

Further examples representing extensions to the base system necessary to load a variety of models are presented in Sections 3.2, 3.3 and 3.4.

In the final section a normal and colour compression algorithm is presented to enable our Octree Interaction Engine to visualize larger models.

3.1.1 Octree background

Spatial data structures such as the octree have long been useful for tasks such as efficiently finding the closest point to a given location and other distance queries (see Samet [Sam90]). The way in which an octree is built can have a critical affect on their application and robustness. One classical way of creating an octree in main memory, and indeed of creating other space division trees such as a BSP tree, involves creating and deleting temporary memory to reassign triangles to child nodes. This can easily cause a system to page when subdividing the root of a large scene. One widely used method of octree construction in out of core methods is to scan all the vertices of a model to find its extent, and then create files of either leaf node triangles [CRMS03] or of fundamental quadrics of each octnode [LS01b] and perform several external memory sorts on vertices or quadrics respectively.

We present in the next section a memory friendly way of creating an octree that does not create temporary memory for the triangles, and does not store triangles at leaf nodes. We use the fact that octnodes are defined uniquely in 3D space to allow in-place sorting of triangles

according to their 1 to 8 id-tags. This approach requires only the start and end of the triangle indices in each octnode to be recorded. This idea is easily extended to other space partition trees such as the quadtree provided that the partitions are uniquely bound in space.

We use the word *octnode* or *nodes* to refer to the data structure elements that make up the octree, the words *octree volumes* to refer to the volumes encompassed by octnodes, the word *sub-space* to refer to a three dimensional space enclosed within an octnode and finally *rendernodes* to refer to the data structure elements that make up the RenderArray, which are essentially pointers to octnodes.

3.1.2 Octree construction

Several octree implementations start by creating a duplicate array or list of the chosen geometric primitive of the original model. This array is then either reset or destroyed as the geometry is passed to new lists in child nodes. Unfortunately, this characteristic can undermine algorithms that would otherwise be robust as the memory requirements for creating a duplicate array of the size of a large model can cause a system to page and slow down dramatically. Our approach does not create a duplicate array. Instead it directly uses the object's global array of triangle primitives. Our root node of the octree has two triangle index numbers, the first records the starting position of the first triangle in the global array, and the second the end position of the last triangle. It is then possible to find the maximum extent of the object and to subdivide space regularly at half distances. At each subdivision step there are three phases, a counting phase (O(N)), a sorting phase (O(N*log(N))), and a node creation phase (O(N)) (see Algorithm 1).

Phase one - counting: The first phase of the subdivision process is a single pass on the array of triangles in the range defined by the node's two index numbers to test in which of the 8 subspaces each individual triangle is contained. On determining where a triangle is contained the triangle is marked with a numeric tag between one and eight corresponding to the lexicographical order of the subspaces. Note that Cignoni et al. [CRMS03] tag vertices and write them to leaf addresses whereas in this case triangles are tagged and memory sorting of one-dimensional tags is performed. At the same time a counter for each of the eight subspaces is incremented according to how many triangles were contained in each subspace. The centroid of each triangle is used to determine if a triangle is contained in a subspace. The subspaces are thus unique in space.

Phase two - in place sorting of partial id: The next phase is to sort the triangles in the range of the node. Sorting is carried out within the original data structure according to their given tag ids. It is important to note that, since the child nodes are mutually exclusively contained within the parent octnode's 3D volume, sorting within the child nodes does not affect the global order

of the triangles with respect to higher nodes.

Phase three - sub node creation: In the third and final phase, each counter is sequentially checked, creating new sub-nodes if a counter is non-zero. The structure is recursively subdivided until either a threshold determining the maximum number of triangles the child nodes may contain (MAXTRI) or a maximum depth threshold (MAXL) is reached. Provided the node contains triangles, the starting index of the first node is the same as the starting index of the root node, whilst the ending index of the node is the starting index plus the number of triangles encountered in that node. Node indices progress seqentially with the starting index of the second node being equal to the number of triangles contained in the first node. Sub-nodes are created with the same procedure as outlined in pseudo-code in Algorithm 1.

Algorithm 1 Pseudo-code for octree subdivision.

```
function Subdivide (thenode) {
if((level<MAXL)&(thenode->end-thenode->start>=MAXTRI)) {
    o1count = 0 o2count = 0 o3count = 0 ... o8count = 0
    // PHASE ONE - COUNTING
    for(i=thenode->start i<=thenode->end i++) {
        onode=0 face=atFaceArray(i) p=face->calculate_centroid()
        if((p.X()>=thenode->X())&(p.Y()>=thenode->Y())&
        &(p.Z()<=thenode->Z()))          {o1count++ onode=1}
        elseif((p.X()>=thenode->X())&(p.Y()>=thenode->Y())&
        &(p.Z()>thenode->Z()))           {o2count++ onode=2}
        elseif((p.X()<thenode->X())&(p.Y()>=thenode->Y())&
        &(p.Z()<=thenode->Z()))          {o3count++ onode=3}
        elseif((p.X()<thenode->X())&(p.Y()>=thenode->Y())&
        &(p.Z()>thenode->Z()))           {o4count++ onode=4}
        elseif((p.X()>=thenode->X())&(p.Y()<thenode->Y())&
        &(p.Z()<=thenode->Z()))          {o5count++ onode=5}
        elseif((p.X()>=thenode->X())&(p.Y()<thenode->Y())&
        &(p.Z()>thenode->Z()))           {o6count++ onode=6}
        elseif((p.X()<thenode->X())&(p.Y()<thenode->Y())&
        &(p.Z()<=thenode->Z()))          {o7count++ onode=7}
        else /*((p.X()<thenode->X())&(p.Y()<thenode->Y())&
        &(p.Z()>thenode->Z())) */        {o8count++ onode=8}
        if(onode!=0){//optionally use below for global indexing
        //face->set_groupid(face->get_groupid()*10+onode)
            face->set_groupid(onode) }
        // PHASE TWO - SORTING
        st=thenode->get_startf()
        numFaces=thenode->get_endf()-thenode->get_startf()+1
        sort(st, st+numFaces, groupid_compare())
        // PHASE THREE - SUBNODE CREATION
        tmpstart=thenodes->get_startf()
        if(o1count>0){
        tmp=newNODE(thenode,1,tmpstart,tmpstart+o1count-1)
        thenode->set_o1(tmp); tmpstart=tmpstart+o1count}
        if(o2count>0){
        tmp=newNODE(thenode,2,tmpstart,tmpstart+o2count-1)
        thenode->set_o2(tmp); tmpstart=tmpstart+o2count}
        ---
        if(thenode->get_o1()!=0){    //subdivide further
        level++ subdivide(thenode->get_o1() level--}
        if(thenode->get_o2()!=0){
        level++ subdivide(thenode->get_o2() level--}
        ---
}}}
```

Table 3.1 shows memory and performance results of various aspects of the Octree Interaction Engine construction phase. The rightmost column demonstrates that the overall memory required for the octree is small, fulfilling our aim of keeping both the model and visualization data structures in main memory. The fourth column shows that the octree can be built rapidly enough upon reading a model. This performance result can be attributed to the in-place sorting of geometric primitives, instead of resorting to temporary memory allocation and deletion of geometry and leaves as in the classical approach to octree construction. The time given for octree construction also includes creation of a clustered navigation skin of the size reported in the second column. The time for clustering the model is below 10 seconds for all models when using a G4-500Mhz PowerPC processor laptop computer. It should be noted that the models of the two bottom-most rows are comprised of multiple surfaces or multiple objects and are visualized without a navigation skin.

model name	model # {triangles / vertices}	navigation skin # {triangles / vertices}	time(s) build octree	time(s) file read	octree max. {depth / verts}	# octree nodes {leafs / total}	octree memory (MB)
Thai statue	10,000,000 / 4,999,996	8,606 / 3,458	172	49	10 / 100	287,951 / 357,679	21.4
Lucy statue	28,055,742 / 14,027,872	9,067 / 3,877	543	150	10 / 100	1,013,687 / 1,256,413	75.3
Skull	1,609,594 / 1,234,798	2,224 / 312	38.2	35	12 / 100	49,042 / 60,946	3.9
Canoptic chest	284,399 / 713,760	412 / 210	5.3	34.5	8 / 100	7,525 / 9,666	0.6
Batalha -entrance-	1,160,186 / 597,118	525 / 251	44.7	85.4	16 / 100	34,219 / 44,078	2.9
UNC power plant	12,748,510 / 11,070,509	- / -	221	8331	12 / 1,000	44,560 / 53,170	3.1
Brain	1,142,182 / 571,095	- / -	12	6	10 / 10	286,120 / 349,526	20.9

Table 3.1: Octree Interaction Engine performance and memory requirements. The 4th and 8th column from the left show that the octree construction and memory usage is both robust and compact with large models.

The next section presents the RenderArray data structure that maintains pointers to octree volumes to render the triangle budget.

3.1.3 RenderArray

Once the scene has been represented hierarchically in the octree an analogous structure to the *active front*[1] of a vertex tree can be built. The RenderArray is an array comprised of an appropriate number of pointers to octnode volumes (rendernodes) that can be sorted fast enough and the required distances computed fast enough for interactive rendering of large models.

The RenderArray essentially consists of an array of pointers to octnodes with a variable RMAXTRI, similar to the threshold MAXTRI used in the octree construction phase (Section 3.1.2). The RenderArray is created by checking the triangle index range of the rootnode. If there are more triangles than RMAXTRI, then the node pointer is not stored in our RenderArray but the sub-nodes are accessed instead and a node inserted into the array only when the range is less than or equal to RMAXTRI. Initially, RMAXTRI is set to the same value as MAXTRI so that the resulting RenderArray will have as many elements as there are leaf nodes. Figure 3.2 shows the octnode volume(s) of different length RenderArrays of increasing spatial accuracy being rendered from left to right with a fixed rendering triangle budget of 4,000. The octnodes in blue represent volumes within which all triangles have been rendered within the triangle budget. The octnodes in pink represent volumes where triangles have not all been rendered. When answering the question "which are the closest triangles or closest octnode volume to the intersection point of the line of sight with the model?", the smaller the octnode volumes used in the RenderArray, the more spatially accurate the answer will be. Figure 3.2 demonstrates that as the spatial accuracy of the RenderArray increases towards the right, the blue volumes also appear closer to the surface of the model.

3.1.4 System initialization

A user might have a computer system with a fast central processing unit but a slow graphics card. In order for our system to be able to adapt to available resources the rendering time has been separated from computation time. For any given rendered frame, there is a total elapsed time comprised of computation time and rendering time. This separation allows for independent adjustments to the computational load and to the rendering load in order to meet a target frame rate.

Watson et. al [WSNR96] report on a range of frame rate requirements for acceptable user performance in different applications. A frame rate of 10-15 frames per second is often required

[1]This is a data structure that keeps track of what resolution or level in a vertex tree each part of a model is in when rendering, see section 2.4.2.

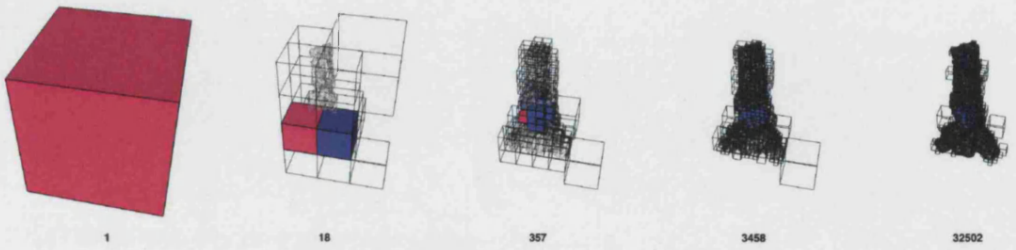


Figure 3.2: RenderArrays with an increasing number of render nodes and spatial accuracy from left to right; computed on the 10 million triangle Thai statue scanned model. All render nodes of the RenderArrays are rendered in wireframe. Render nodes associated with triangles being rendered towards the triangle rendering budget are colour coded as follows: nodes whose triangles are rendered within the 4,000 triangle budget are rendered in blue; nodes whose triangles exceed the triangle budget are pink. Note that in the second sub-image from left, the large render node in blue has small overlap with the model, hence all its triangles are rendered within the triangle budget.

for a fully "acceptable" performance in an immersive virtual environment. Let us temporarily consider only the computational side of rendering a frame. For a single frame the computation process must calculate the intersection point of the line of sight with the model using the octree, compute distances from the intersection point to the four corners and centre point of each octnode identified in the RenderArray, and sort the RenderArray. Given an instantaneous rendering process, the computation process can take at most 0.04 seconds to achieve a 25 frames per second target. Since the computation time increases with the length of the RenderArray (Table 3.2), an appropriate maximum RenderArray length can be determined by iteratively timing the computation time and adjusting the maximum length of the RenderArray until the required time is less than the 0.04 second target. In practice we aimed for a target frame rate of 10 frames per second, limiting the computation time to 0.1 seconds. This threshold automatically determined that the length of RenderArrays that can be processed by the G4-500Mhz PowerPC processor was of the order of a few tens of thousand of nodes (highlighted in bold in Table 3.2 is the length of the RenderArray chosen by the system for each model). It can be seen that initially the RenderArray has the same number of render nodes as there are leaf nodes (the first column of the table is identical to the second column from right of Table 3.1) and that the system adjusts for larger models by choosing RenderArrays with fewer nodes than there are leaf nodes.

Further adjustments are possible at run-time where the user can elect to create a shorter and faster RenderArray.

model name	**RenderArray # render nodes / sort time (s)**			
Thai statue	287,951 / 0.51	**32,502** / 0.048	3,458 / 0.0034	357 / 0.0004
Lucy statue	1,013,687 / 2.2	76,279 / 0.14	**8,898** / 0.014	874 / 0.0006
Skull	**49,042** / 0.07	4,693 / 0.004	623 / 0.0004	60 / 0.0002
Canoptic chest	**7,525** / 0.007	689 / 0.0005	99 / 0.00008	-
Batalha entrance	**34,219** / 0.04	3,327 / 0.004	272 / 0.002	-
UNC power plant	**44,560** / 0.05	5,067 / 0.05	548 / 0.0003	-
Full brain	286,120 / 0.63	**39,867** / 0.07	4,900 / 0.0047	464 / 0.0003

Table 3.2: RenderArray size and performance. Highlighted in bold is the length of the RenderArray automatically chosen by our system to meet a computation time of 0.1 seconds for a target of 10 frames per second.

The first RenderArray created by the system has the same number of elements as there are leafnodes in the octree (Section 3.1.3). The system can therefore default RMAXTRI to the threshold MAXTRI used in the octree at start-up time.

The system then times how long it takes to complete one pass of computing the distances of the RenderArray node corners and centres to a 3D point and to sort the RenderArray according to the smallest of these distances. If the system takes more than a default time of 0.1s, the system destroys the RenderArray and creates a new one (Section 3.1.3) with RMAXTRI increased by a factor of ten in our implementation, giving fewer nodes for which computations must be made at the expense of loss of spatial accuracy.

An interesting observation can be made from Figure 3.3. In the upper left image, the RenderArray depth is zero resulting in just one node to be "sorted" and for which a distance must be computed. The triangle budget is spent on the first triangles of the RenderArray (in this case the rootnode). These triangles are far from the foremost intersection point of the line of sight with the model. Although our mesh gains coherence through the triangle reordering during the octree construction phase rendering triangles in this manner is like rendering the first triangles of any un-organized triangle soup. As the depth of the RenderArray is increased, for example by dividing RMAXTRI by ten, more nodes require sorting and computation but the triangle budget is spent on nodes that are closer to the intersection point with the line of sight. As may be seen in the second image from the left in the upper row of Figure 3.3, increasing the depth of the RenderArray from four to five adds little visual benefit whilst requiring more computation. At depth four, the triangles retrieved from the nodes are already spatially accurate enough and the triangle budget is successfully spent mostly near the foremost intersection point of the line of sight with the model.

The next section describes how the RenderArray is used in the render loop.

3.1.5 Display

Once a RenderArray has been created it is retained and used for rendering any frame unless the user chooses to make a smaller or larger RenderArray by changing for example RMAXTRI in multiples of ten. For each frame being rendered it is necessary first to determine the foremost intersection point of the line of sight with the model using the octree. Whilst traversing the octree, the fast ray rejection test defined by Xu et al. [ZZL03] is used quickly to reject the possibility of finding an intersection point with an irrelevant octnode. A ray is likely to intersect more than one octnode and more than one triangle in the model, so a track is kept of the smallest positive distance from the viewer's position defining the start of the ray and each of the planes defined by the bounding box of the octnodes. This allows one to navigate within a volume.

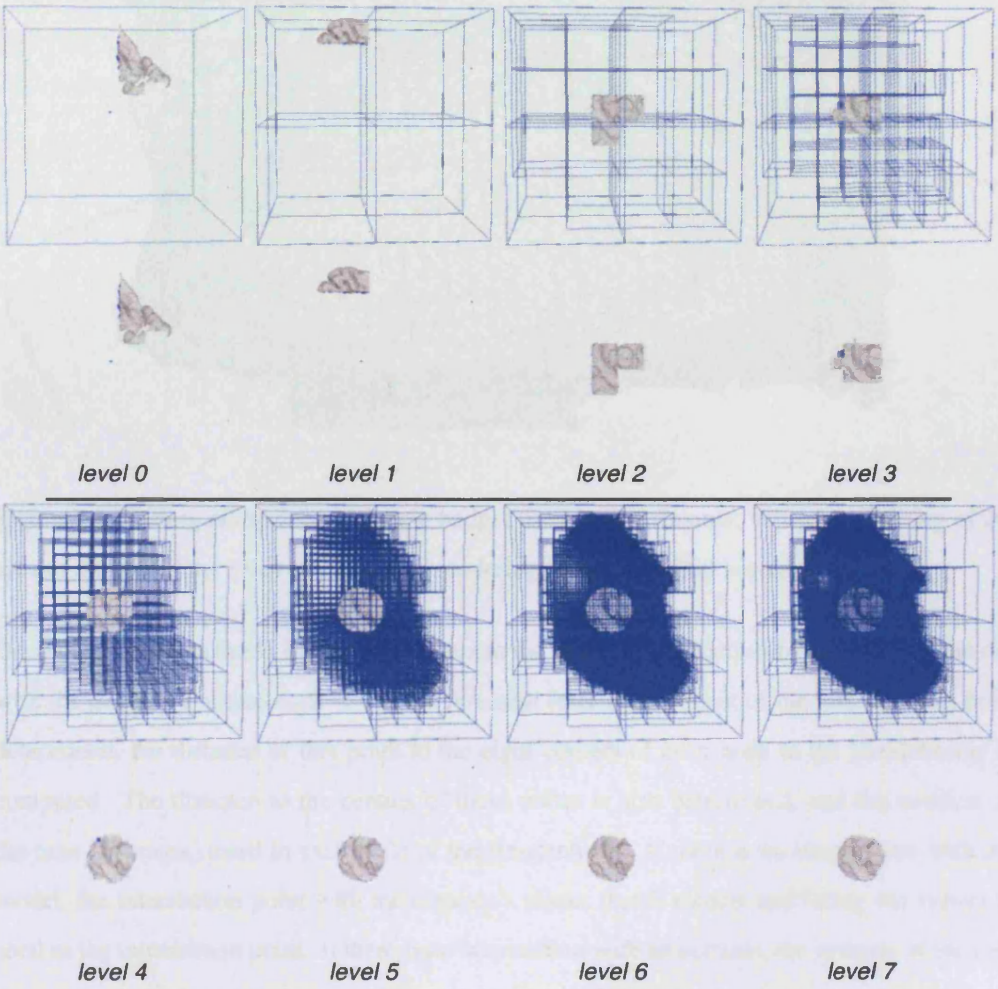


Figure 3.3: The top left image (level 0) shows that the resulting first 16,128 triangles of the octree's root are spatially coherent and are ideal for mesh streaming (Isenberg et al.. [IL05]). From left to right, top row to bottom row the lower sub-image of each level shows renderings of a fixed 16,128 triangle budget with RenderArrays of 1 node; 8, 251, 2,708, 20,814, 20,814, 20,814 and 20,814 nodes respectively. From left to right the top sub-images of each level show the octree being rendered at depths 0 to 7.

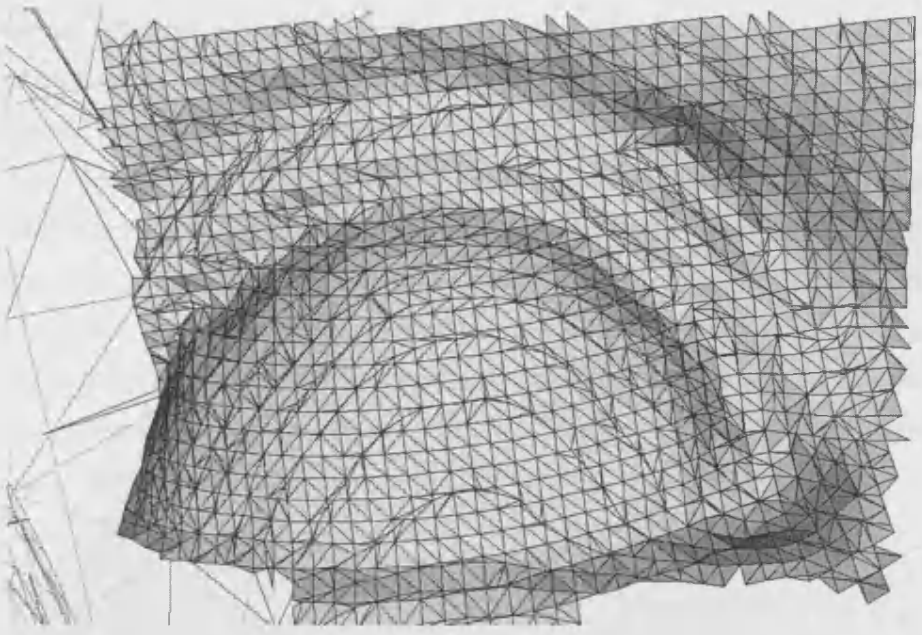


Figure 3.4: Editing distance and triangle budget used in experiments: full screen editing of the area of the eye of the Lucy model with a rendering budget of 4,000 triangles.

Once the relevant octnode is found we compute the point of intersection by intersecting the ray with the geometry at that node. Once the foremost intersection point of the line sight has been determined, the distance of this point to the eight corners of each node in the RenderArray is computed. The distance to the centres of these nodes is also determined, and the smallest of the nine distances stored in each node of the RenderArray. If there is no intersection with the model, the intersection point with the octnode's planes that is closest and facing the viewer is used as the intersection point. If there is no intersection with an octnode, the octnode of the root whose centre is closest to the viewer is used as a proxy for the intersection point.

Finally, the RenderArray is sorted according to the stored distances and rendered with a triangle budget set by the user. Rendering is accomplished by first rendering every triangle in the first RenderArray octnode and then rendering triangles from the next octnodes in the RenderArray until the triangle budget is exhausted. We note that in order for a user to be able to select an individual triangle from a scanned model, the user has to be at relatively close distance to the model to be able to visually distinguish the projected triangle edges. Results demonstrated that a triangle budget of 4,000 was sufficient to fill the screen at an editing distance as can be seen in Figure 3.4 when editing the area of the eye of the 28 million triangle model of Lucy. Accordingly a triangle budget of 4,000 is used for these experiments.

As mentioned in the beginning of the chapter, different types of objects have different

intrinsic modelling properties. The base system allows the user to choose the octree depth at which to render octnodes in wireframe. This rendering mode is particularly useful when editing multiple surface objects such as the closely paired grey and white matter surfaces of the human brain. With multiple surfaces simple clustering can create disturbing self-intersections of the coarse overview model. Figure 3.5 shows on the left a triangle budget of 4,000 being rendered with a RenderArray of 4,900 nodes and octnode box rendering of depths 3 to 5 from the top downwards. On the right a full 1.1 million triangle budget is used with the same octnode boxes. The model in Figure 3.5 is the final result obtained from editing the 596,872 triangle model of the left hemisphere of the human brain shown in Figure 1.3. The editing task is described in detail in the next section.

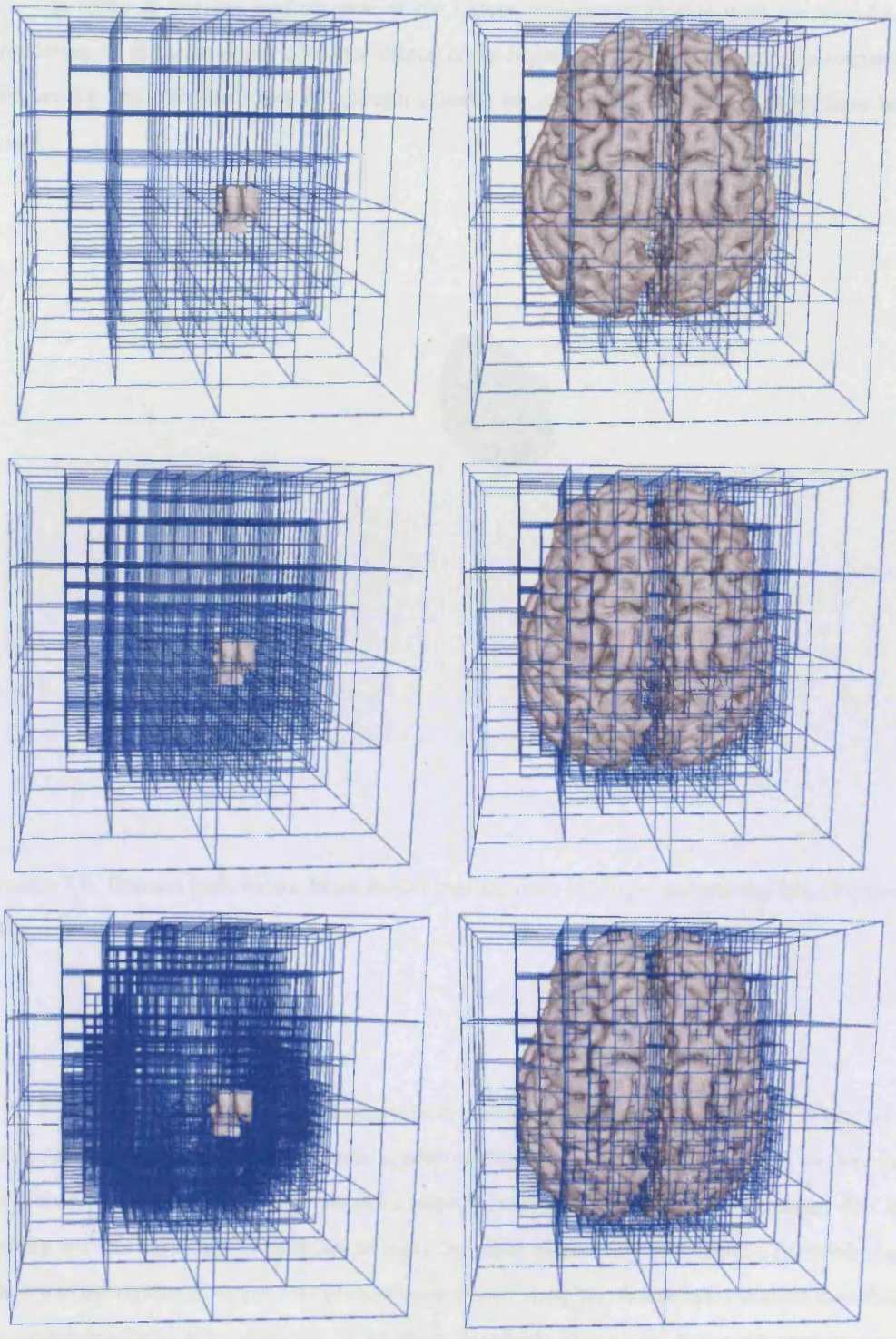


Figure 3.5: Octree depth rendering using a fixed size RenderArray of 4,900 nodes, from top to bottom: depth 3, 4 and 5; *left:* using a 4,000 triangle rendering budget; *right:* full 1,142,182 triangle budget.

In order to test the performance of the Octree Interaction Engine with the wire-frame rendering of different depth octnodes displayed in Figure 3.5, the camera path parameters of Figure 3.6 and a RenderArray of a length suitable for interaction with 4,900 nodes have been used.

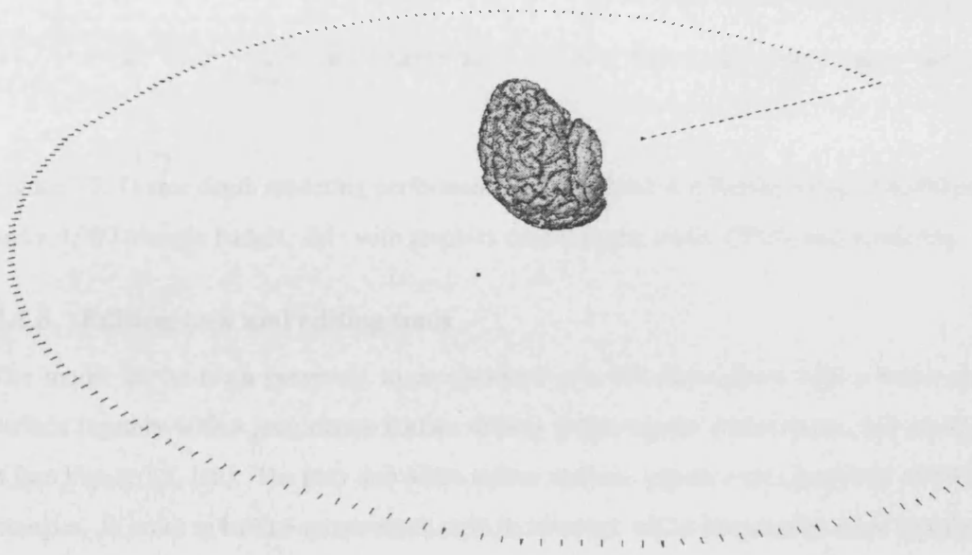


Figure 3.6: Camera path for the brain model starting from the right and moving in a clockwise direction with respect to the model.

Figure 3.7 shows that a frame rate between 16 and just under 12 is possible at any point of the camera path with the wire-frame rendering of octnodes at a depth of 3. It is evident that as the user becomes closer to the model a point is reached at which the fine geometry fills the screen and no wire-frame rendering is required. Also noteworthy is that the CPU renderings shown in the rightmost figure for the same camera path show less frame rate variation than those using the graphics card on the left. Throughout this thesis all reported frame rate performances with a display window size of 600x600 pixels refer to the rendering results produced by the ATI Rage Mobility 128 graphics card with 8 Mbytes of SDRAM video memory whilst references to a display window size of 1142x718 refer to CPU based rendering of the G4-500Mhz PowerPC processor without graphics card acceleration unless otherwise stated.

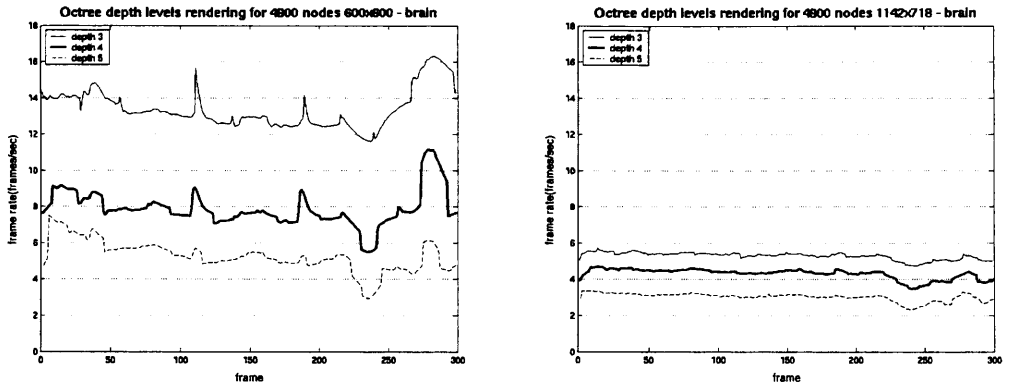


Figure 3.7: Octree depth rendering performance using a fixed size RenderArray of 4,900 nodes and a 4,000 triangle budget; *left:* with graphics card support; *right:* CPU based rendering.

3.1.6 Editing task and editing tools

The model of the brain presented to us consisted of a left hemisphere with a white matter surface together with a grey matter surface closely following the white matter, but exterior to it (see Figure 1.3, left). The grey and white matter surfaces together are comprised of 596,872 triangles. In order to build a symmetrical right hemisphere whilst keeping the same topological genus, the following steps were taken:

1. Images such as the one in Figure 1.3 (middle) were used to plan the perimeter of the area to be removed from the side of the grey matter.

2. The resulting hole boundary was extruded into the mid-brain vertical plane (see red triangles, Figure 1.3, right) and the model mirrored on this plane, defined to be the YZ plane of the model.

3. Vertices on the YZ plane were re-used on the right hand side of the model in order to keep the mesh connected.

In practice, these steps were more easily performed first on the white matter (see extruded triangles in blue passing through the grey matter, Figure 1.3, right) then on the grey matter to selected red triangles around the dark blue extrusion.

In total 25,781 triangles had to be selected and removed, subject to the constraints of no-self intersection or intersection between the two surfaces. In addition, features such as deep chasms on either surface shown in Figure 1.3 (right) required careful 3D inspection and planning. The resulting model of 1,142,182 triangles can be seen in Figure 3.5. The complete edit took around 4 hours.

The algorithm used to select triangles for editing represents a ray as two non-parallel planes passing through the origin as defined by Xu et al. [ZZL03]. Although the triangle picking algorithm is fast, selecting thousands of triangles by hand would be quite time consuming and prone to error. To assist the user, the following tools were created:

1. A selection tool that allows the user to define a radius threshold which is used to tag and highlight all triangles connected to a selected triangle to the depth set in a breadth first traversal of the connected geometry.

2. A purge button that enables the user to de-select small triangle groups and to retain only the largest connected group of tagged triangles.

3. An undo select button.

By way of example, the first tool was used with large radius thresholds to select triangles that were otherwise difficult to access, a radius threshold of 10,000 being particularly useful in the selection of all triangles inside the perimeter of the areas to be removed from the side of the model. In the next section the mesh streaming properties of the mesh after the creation of the octree are described.

3.1.7 Mesh streaming

Cignoni et al. [CRMS03] re-label vertex indices so that they belong to a range of indices defined in the octnode in which the vertices are contained. In contrast, our octree reorders only the triangles and keeps track of the number of sorted triangles within each octnode. Reordering of the triangles in this manner results in a more compact mesh well suited to streaming and compression (Figure 3.5). During mesh streaming, a portion of a model is transmitted progressively. This portion is referred to as the active front. Vertices in this front which require connected vertices to be transmitted are called the active vertices and the number of active vertices is called the front width. The maximal index difference of vertices on the front is called the span.

Quantitatively the result of our octree construction method compares well with the spectral sequencing or single axis vertex sorting of Isenberg et al. [IL05]. For example, the Stanford dragon (Figure 6.32, left) initially has a front width of 1.05% which, after our triangle reordering, reduces to 0.58% compared with 0.18% (for spectral sequencing).

The original dragon had some initial coherence. Models that are largely incoherent from the start such as the Stanford bunny at 26.22% gave 1.9% after our triangle re-ordering compared to 0.78% with spectral sequencing. Further work is planned to re-label the vertices, within each octnode, for example as in Cignoni et al. [CRMS03] and to improve on the front span of the re-ordered models.

3.2 Octree Interaction Engine for scanned objects

In the previous section the base system of the Octree Interaction Engine and a means to visualize multiple surface objects through the wireframe rendering of octnode boxes at a level chosen by the user was presented. In this section an extension for the visualization of scanned objects is presented. The hypothesis that it is possible to create and maintain similar high frame rates with any size model that fits in main memory from any angle is also tested.

Unlike multiple surface objects, scanned objects can be clustered to produce a coherent surface. After the octree is created for an object a navigation skin is created that provides an overview of the extent of the object. This simple mesh is kept in memory and rendered every frame using our depth buffer strategy. This strategy, used in all extensions of the base system, ensures that the fine geometry is always rendered over the navigation skin so as not to obstruct the user's selection of triangles (Section 3.3).

The navigation skin is created using a high quality clustering algorithm [Lin00]. This method adds together all the fundamental quadrics inside an octnode volume, we then find a vertex position that best approximates the triangles within each volume.

A detailed review of the quadric error can be found in Section 2.6. The navigation skin is limited to a few thousand triangles. Since each octnode volume or cluster node represents one vertex in the simplified mesh/navigation skin, the total number of vertices can be determined by choosing an appropriate octree depth in a similiar manner to that used to determine the length of the RenderArray (Section 3.1.3). Once the optimal vertex positions are found each original triangle in the model is evaluated and its vertex ids replaced with the new clustered vertex ids that contain the vertex. The octree was built using the centroid of triangles. In the situation that a vertex of a triangle is not in any of the cluster nodes the triangle's centroid position is used to look-up the cluster node for that vertex. If a triangle were to have two or three vertices within the same cluster node, the triangle would not be inserted in the navigation skin, otherwise it would be inserted.

Figure 3.8 shows the navigation skin of two models clustered on the fly. The model on the left is comprised of 10 million triangles whilst the one on the right is comprised of 28,055,742 triangles. It was only possible to read these models into main memory by using PNORMS, our colour and normal compression algorithm presented later in Section 3.5. The same section demonstrates that there is no time penalty in the decoding of normals or colour attributes.

Figure 3.8: *Left:* Thai statue model of 10 million triangles and its clustered navigation skin of 8,606 triangles; *right:* Lucy statue model of 28 million triangles and its clustered navigation skin of 3,877 triangles.

Figure 3.9 demonstrates how a user can control the size of the RenderArray at run time for more spatial accuracy.

Figure 3.9: Rendering budget of 4,000 triangles using RenderArrays of increasing size and spatial accuracy for the Lucy model; from left to right 874, 8,898, 76,279 and 1,013,687 render nodes respectively.

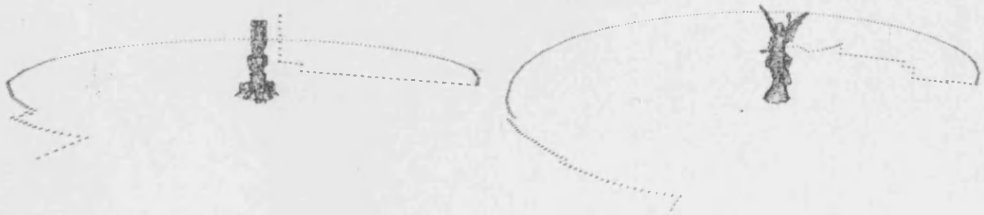


Figure 3.10: Camera path for the Thai statue (left) and for the Lucy statue (right) starting from the left and moving in a clockwise direction with respect to each model.

In order to test the Octree Interaction Engine with these two models, one of which has almost three times the triangle count of the other, similar camera paths for both were created (Figure 3.10). Figure 3.11 demonstrates that a similar rendering performance is possible with both models when a RenderArray of similar length is used. In particular, if a RenderArray of 8,898 nodes is used for the 28 million triangle model and a RenderArray of 3,458 nodes is used for the 10 million triangle model, a frame rate above 14-15 frames per second can be maintained throughout the camera path.

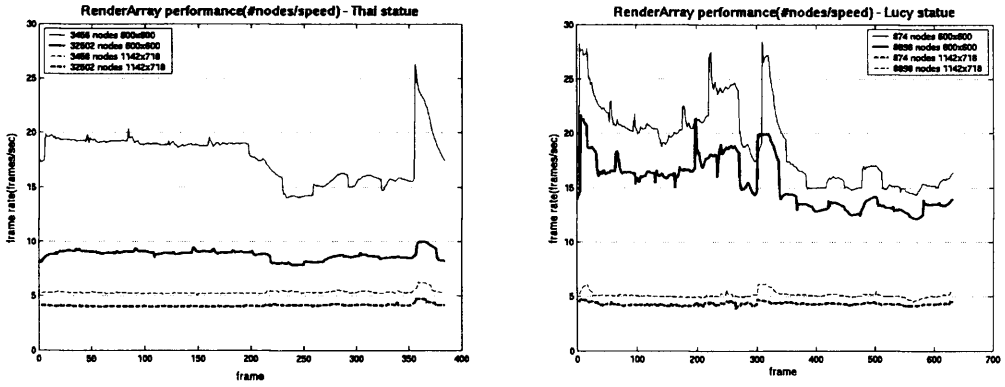


Figure 3.11: Similar RenderArray rendering performance for the 28 million triangle model using 8,898 render nodes (right) and the 10 million triangle model using 3,458 render nodes (left).

Figure 3.12 reports the difference in frame rate with larger triangle budgets set by the user for the same camera path for the 10 million triangle model. In the figure on the right it can be seen that the variation of the frame rate is smoother with CPU based rendering than when a graphics card is used.

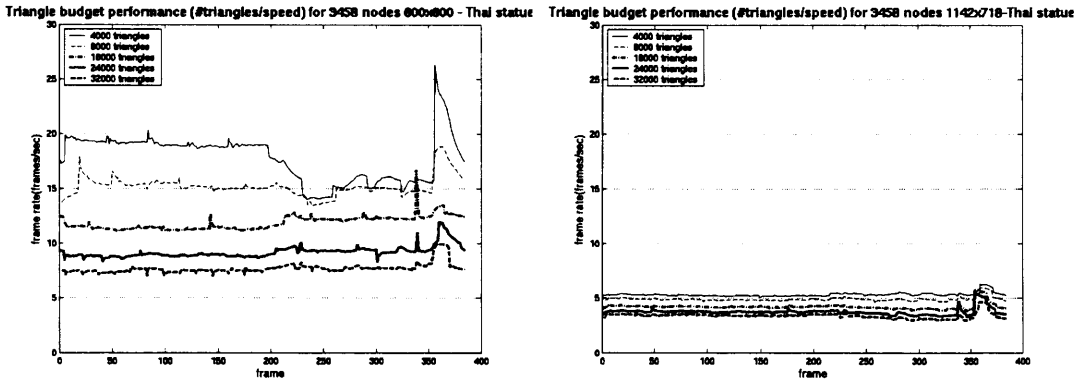


Figure 3.12: Triangle budget rendering performance using a fixed size RenderArray with graphics card support (left) and with CPU based rendering (right).

As expected from the manufacturer's literature, more powerful graphics cards support larger triangle budgets at a given frame rate. For example, if a graphics card specification[2] states that it can render 4.5 million triangles per second then it can render a triangle budget of 284,557 triangles at 15 frames per second. A user can however elect to temporarily increase the triangle budget in order to explore the model in detail (Figure 3.13). An idle function can progressively render larger budgets whilst the user's viewpoint is stationary. In such situations asynchronous rendering becomes useful as it can pre-empt the rendering of a large triangle budget when the user starts moving again thereby ensuring interactivity by reverting to the smaller triangle budget previously set.

[2]The benchmarks used in specifications vary considerably. Some refer to the rendering of pixel-sized triangles. Such bechmarks are however beyond the scope of this thesis.

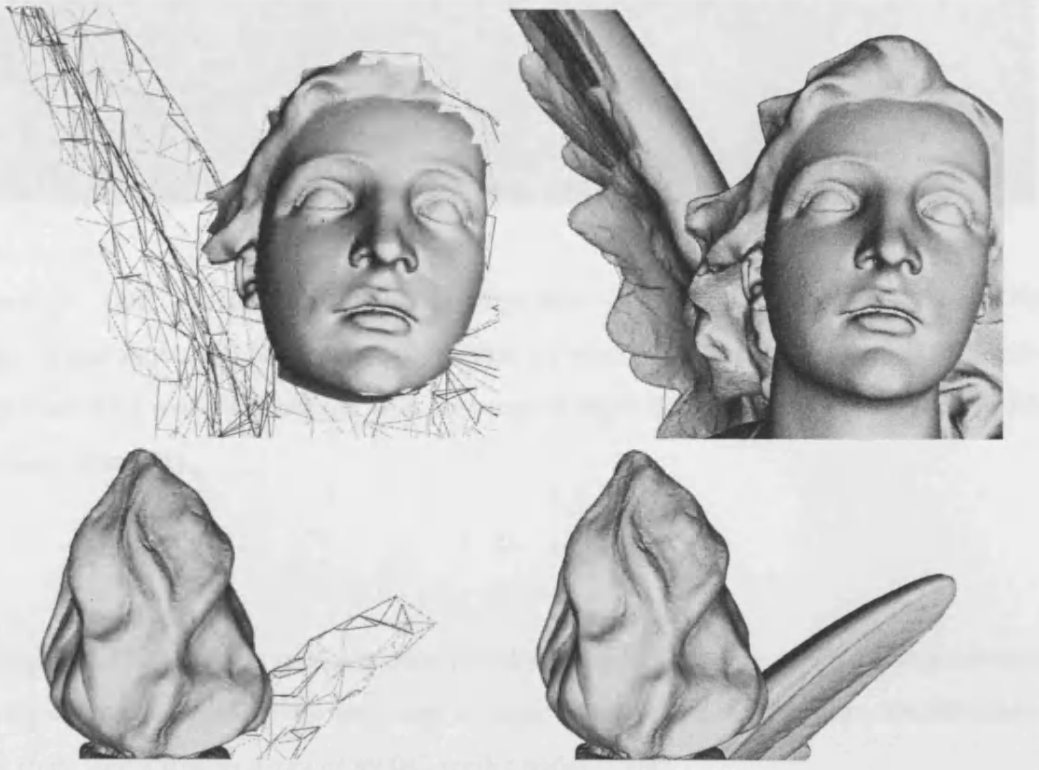


Figure 3.13: Progressive rendering - *left:* 284,557 triangle budget; *right:* full 28,055,742 triangle budget.

The following section demonstrates by example, that the Octree Interaction Engine is useful in the context of viewing museum scanned artifacts.

3.2.0.1 Vertex colour - museum colour scanned artefact

Figure 3.14 shows a clustered navigation skin of 2,224 triangles extracted from a scanned skull model comprising of 1,609,594 triangles and 1,234,798 colour vertices. The second and fourth image from the left show renderings using our depth buffer strategy developed in this thesis and presented in the next section. The method ensures that the fine geometry is always rendered over the navigation skin.

Figure 3.14: LoD occlusion: Coarse Navigation skin in wireframe rendering occluding the triangle rendering budget and hindering the user's triangle selection task (first and third sub-image from left); without occlusion with pre-emptive depth buffer strategy (second and fourth sub-image from left).

Figure 3.15 shows the spatial accuracy trade-off when rendering a 4,000 triangle budget from the intersection point of the line of sight and the model with a RenderArray of 4,693 render nodes (left), and a RenderArray of 49,042 render nodes (right).

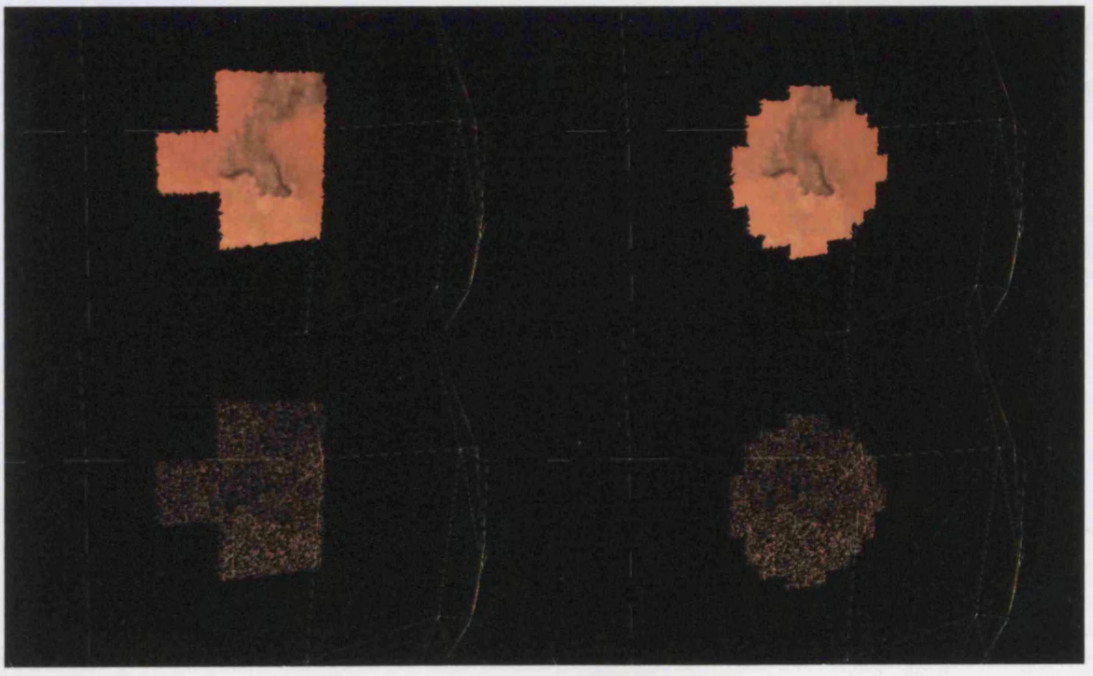


Figure 3.15: Octree Interaction Engine rendering of a museum colour scanned type of object with vertex colour interpolation. Rendering budget of 4,000 triangles using RenderArrays of increasing size and spatial accuracy for the vertex colour scan of the Skull; *left:* 4,693 render nodes; *right:* 49,042 render nodes.

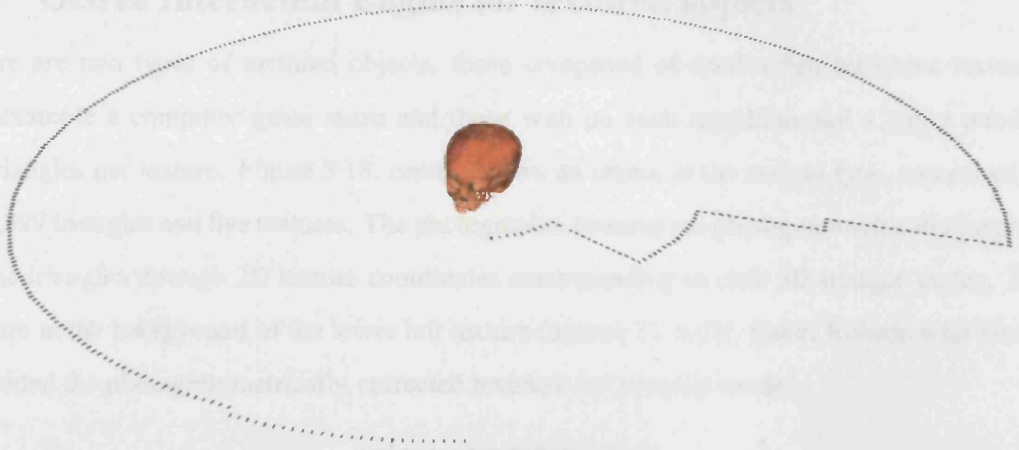


Figure 3.16: Camera path for the Skull model, starting from the bottom and moving in a clockwise direction with respect to the model.

The performance of the two RenderArrays was tested with the camera path shown in Figure 3.16. Figure 3.17 demonstrates that a rendering performance above 14-15 frames per second can be achieved in any viewing condition with a RenderArray of 4,693 render nodes.

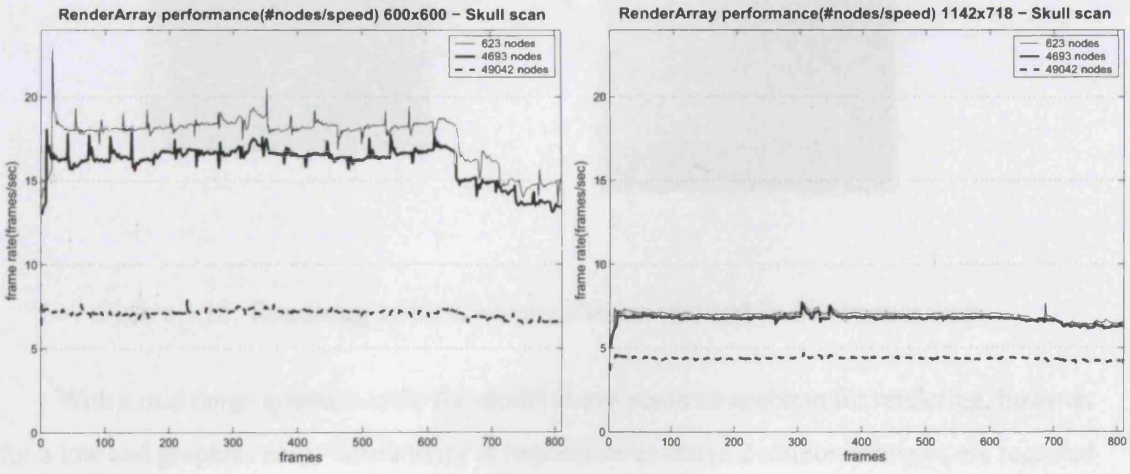


Figure 3.17: Triangle budget rendering performance using a fixed size RenderArray with graphics card support (left) and with CPU based rendering (right).

The next section presents the extensions required to the base system in order to accomo-

date textured objects and presents a depth buffer strategy that ensures that the fine geometry is rendered over the navigation skin.

3.3 Octree Interaction Engine for textured objects

There are two types of textured objects, those composed of small often repetitive textures, for example a computer game maze and those with no such repetition and a larger number of triangles per texture. Figure 3.18, centre, shows an object of the second type, comprised of 284,399 triangles and five textures. The photographic textures are photogrammetrically mapped to the triangles through 2D texture coordinates corresponding to each 3D triangle vertex. The person in the background of the lower left texture (texture 1), is Dr. Stuart Robson who kindly provided the photogrammetrically corrected textures and scanned model.

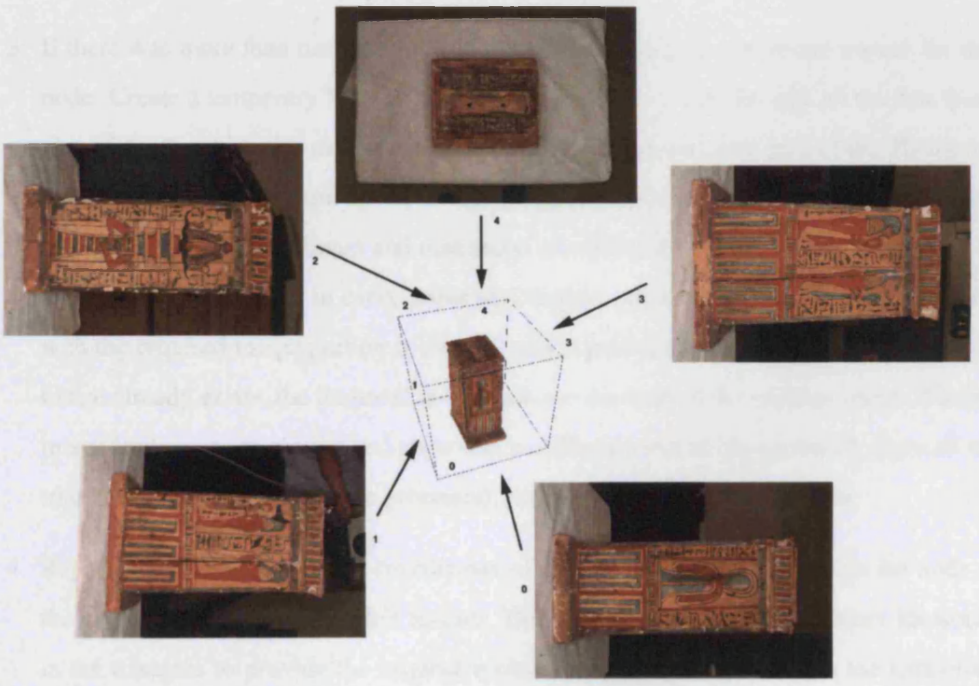


Figure 3.18: Rendering of the Canoptic chest model and its five texture maps.

With a mid range graphics card, the model above poses no problem for rendering, however for a low end graphics card, interactivity is impossible as above 2 million triangles are required to be rendered for a frame rate of for example 10 frames per second, thus greatly exceeding the rendering throughput of the card (later in Figure 3.45 we report an average frame rate of around 6 frames per second with a typical low end graphics card and a model comprised of quarter of the number of geometric primitives). Given any rendering hardware there is always a model that is going to exceed the available rendering capabilities. A larger textured model is

also tested in the end of this section. In order to address this issue a new algorithm was devised in which the rendering budget is limited and new textures are created and switched between triangles in volumes near the intersection point of the line of sight with the model.

Algorithm 2 Node texture creation.

```
function create_cluster_textures (thenode) {
```

1. Create a second temporary RenderArray2 as described in section 3.1.3 with RMAXTRI set to RMAXTRI×10, this ensures the creation of a RenderArray with nodes one level higher than the octree's leaf nodes.

2. Look up each triangle of RenderArray2 nodes in turn, and check whether only one texture id is used in that volume. If only one texture is required assign this texture id to the node and proceed with the next node in RenderArray2 and proceed to Step 2.

3. If there was more than one texture id in the volume, create a new image texture for that node. Create a temporary face list and sort on texture id. Loop through all the first triangles of the face list with the same texture id, tracking the min and max of the 2D texture (uv) coordinates referenced by the triangle vertices. Once a triangle with a different texture id is found, the minimum and maximum uv used and the current dimensions (xy) of the new image is stored in every processed triangle. A new temporary image fragment with the required image portion is created, and copied to the node's texture image. If an image already exists, the fragment is appended to the right of the existing image. Further image fragments are created and appended as different texture ids are found. Once all the triangles in the node have been processed, proceed to Step 4 with this node.

4. Reparameterize all uv texture coordinates of the vertices of the triangles in the node to the new dimensions of the node's texture. This step utilises the original texture ids stored in the triangles to provide the original texture dimensions, together with the temporary information stored at the triangle level in Step 3, such as the min and max uv used, and the dimensions the node's texture had when the image fragments were added to find the absolute new uv coordinates.

5. Once all the nodes of RenderArray2 have been processed, propagate the node texture ids downwards to the leaf nodes of the octree.

}

Essentially new smaller textures are computed for volumes whose triangles index more

than one texture, and the vertex's texture coordinates are reparameterized to index the newly computed textures.

In Step 1 of Algorithm 2, instead of having an image pyramid for textures of the same extent with lower resolutions the method provides textures with smaller extent at the same resolution with a smaller number of triangles indexing them. This new form makes the textures more easily managed and suitable for our Octree Interaction Engine as the spatial extent of the new textures is aligned with the dimensions of the volumes used for rendering the triangle budget from the RenderArray.

Volumes whose triangles index only one texture are unchanged, and the original texture kept (Step 2). Volumes whose triangles index more than one texture, have a new single image texture created by appending the image fragments of the different texture id triangles (Step 3). A temporary auxiliary data structure is built to sort the triangles within the volume according to texture id. Each sequence of triangles with the same texture id creates an image fragment that is appended to the right of the new texture. Before proceeding with a new sequence of texture id triangles, the size of the current new image is stored, and the min max texture coordinates (uv) referenced by the triangles are also stored in every triangle of the sequence. This temporary information allows reparameterization of the texture coordinates for new image texture at the node after all triangle sequences in the node have been processed.

The size of the volumes of the RenderArray is determined as before by timing of the computation load. However textures can be built at a higher level of the octree than the RenderArray, as the triangles within subvolumes are the same triangles that have had their texture coordinates reparameterized higher up (Step 4). For illustration purposes the textures for the model in Figure 3.18 were built exclusively at the first octree subdivision level (Figure 3.19). Each of the resulting sub images are smaller than any of the original images due to the smaller combined triangle area in a volume and because large portions of the original textures that were not used or indexed by any triangle vertex have been deleted. Such deletion can be particularly useful for optimizing the limited memory texture space in the video memory of graphics cards. Additionally, the total number of triangles in the model has been divided into eight smaller and more manageable textured parts ***automatically***. We also note that none of the eight octree volumes index only one of the original five textures, thus the original textures are deleted and the new textures are used instead. In general volume textures are built for volumes one level higher than the RenderArray, allowing the user to temporarily switch the visualization to obtain higher rendering performances (Step 5).

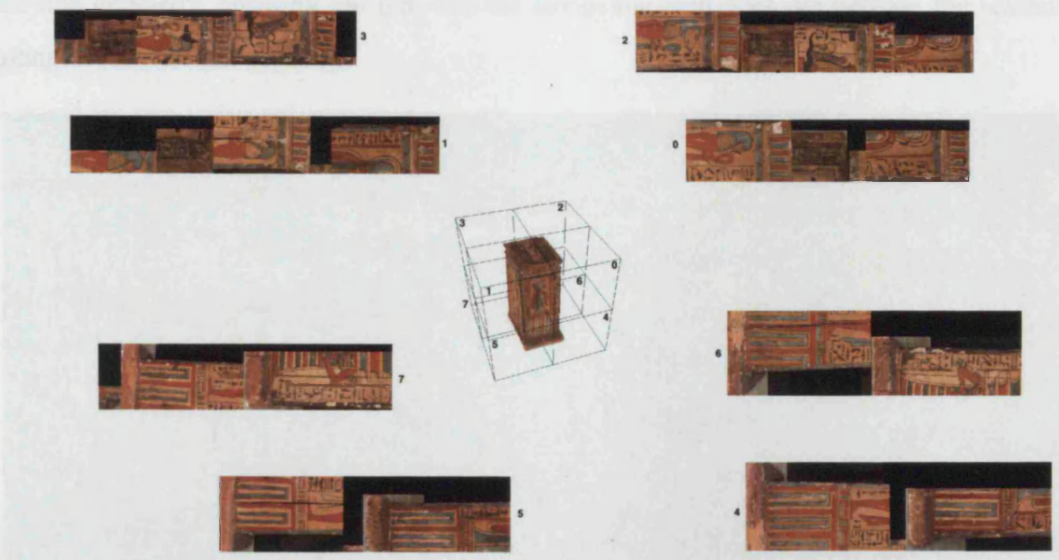


Figure 3.19: Octree Interaction Engine's eight computed textures and full triangle budget rendering of the Canoptic chest model (centre).

The order in which triangles are rendered is typically not sorted in depth as this would add another computation step. Consequently a front facing triangle at the back of the model can be rendered over the pixels of a front facing triangle that is closer to the viewer, creating disturbing artefacts as shown in image a) of Figure 3.20. Depth buffering, or Z-buffering enables the storage of an associated Z-depth value of a pixel being rasterized in a buffer (Z-buffer) of the same dimensions of the image buffer. Typical depth functions are given by $\leq, \geq$ to ensure that the smaller Z-depth pixels prevail in the image buffer. The image at the centre of Figure 3.18 was rendered using depth buffering, hence no artefacts are present. However a problem arises when we have two objects where the Z of each varies relative to the other. This issue is illustrated with the navigation skin rendering of the first and third leftmost subimage of Figure 3.14. As can be seen in sub image c) and d) of Figure 3.20, turning z-buffering on for the navigation skin does not ensure that the fine resolution triangles are rendered over/in place of the pixels of the navigation skin. Using two Z-buffers one for each object and combining the rendering result into one would slow the rendering. Fortunately there is a simple solution. First we render the navigation skin with Z-buffering turned on ($\leq$ ztest) to avoid artefacts from sub image b) of Figure 3.20, then we reset or clear the Z-buffer before rendering the fine resolution triangles

whilst keeping the Z-buffering on ($\leq$ ztest). This strategy ignores what was rendered before the fine geometry, ensuring our aim that the navigation skin does not occlude the selectable triangles (sub images e), f), g)).

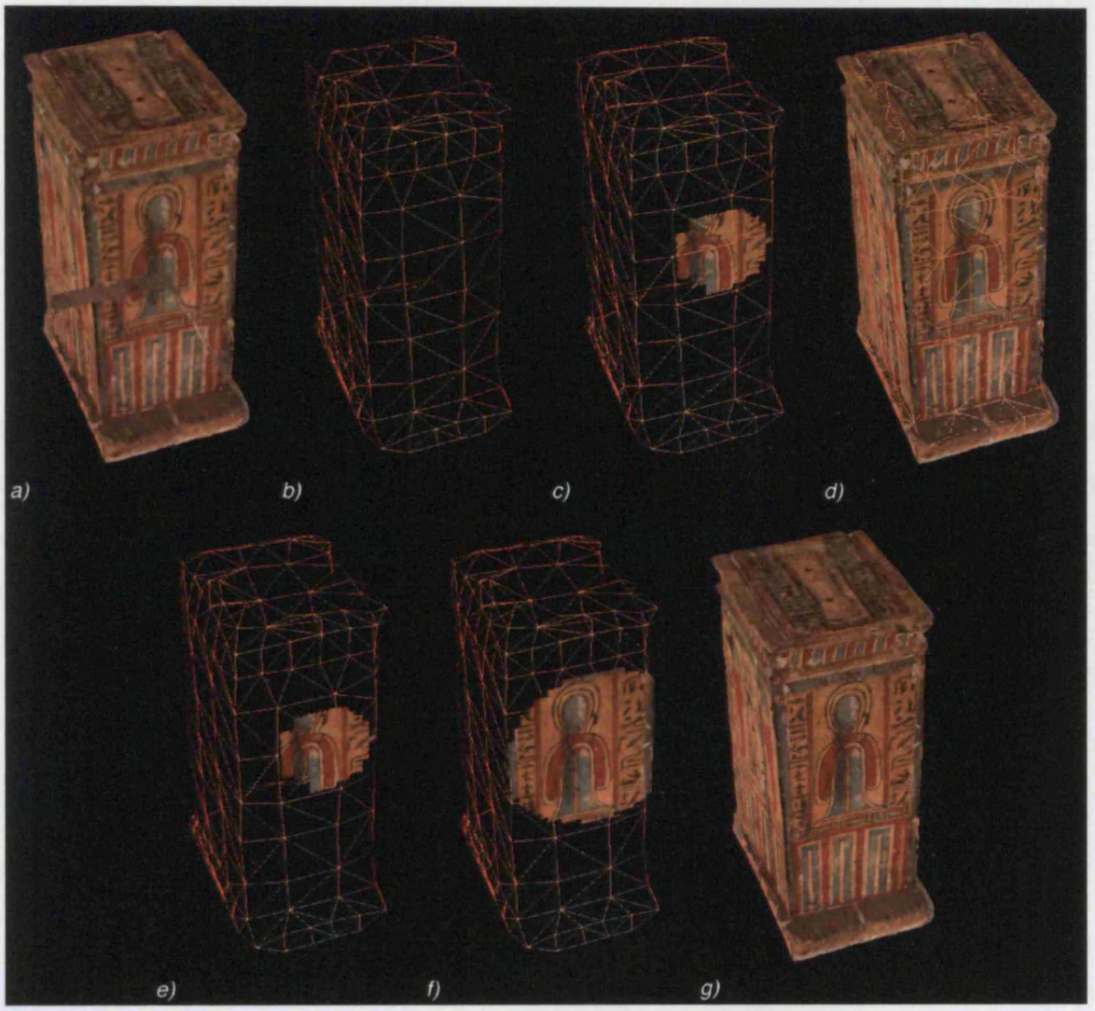


Figure 3.20: Z-buffer strategy - a) front facing artefacts at back of model with depth buffer off and full 284,399 triangle budget rendering; b) minor front facing artefacts at back of model with depth buffer off for navigation skin rendering; c) depth buffering on, navigation skin occludes rendered 4,000 triangle budget; d) same as c) but full budget was used; e) depth buffering on but the Z-buffer was reset after rendering of the navigation skin and before 4,000 triangle budget rendering; f) same as e) but larger budget of approximately 20,000 triangles; g) same as e) but full budget.

The Canoptic chest model has 284,399 triangles, 713,760 vertices, and 5 textures. The created navigation skin has 412 clustered triangles.

Figure 3.21 shows from left to right the spatial accuracy tradeoff when rendering the same

4,000 triangle budget from the intersection point of the line of sight and the model with RenderArrays of 99, 689 and 7,525 render nodes respectively. With a full triangle rendering budget, all the render nodes are used (right most sub image of Figure 3.21).

Figure 3.21: Octree Interaction Engine rendering of a museum colour scanned and textured type of object. Rendering budget of 4,000 triangles using RenderArrays of increasing size and spatial accuracy for the Canoptic chest model; from left to right 99, 689, 7,525 render nodes respectively.

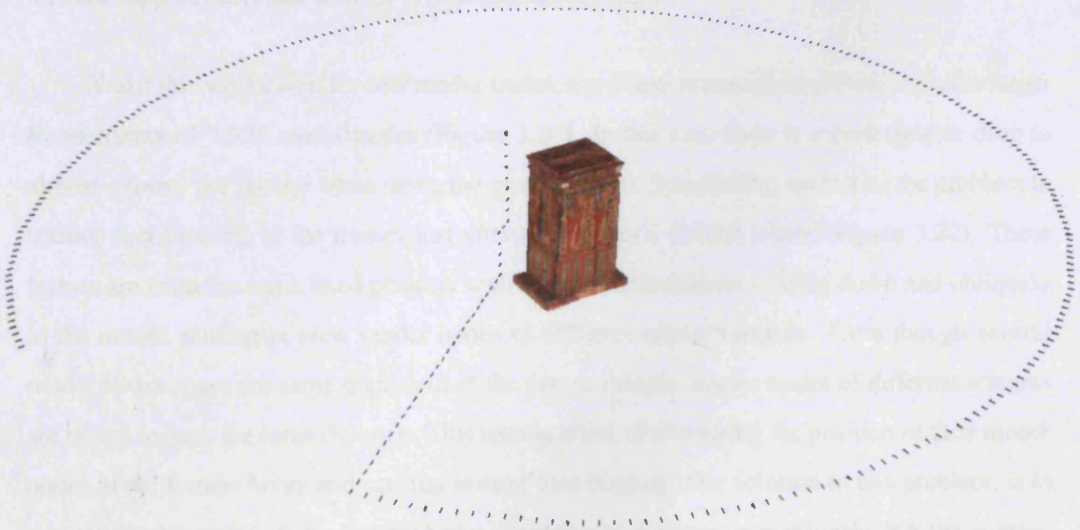


Figure 3.22: Camera path for the Canoptic chest model starting from the left and moving in a clockwise direction with respect to the model.

The performance of the extension of our Octree Interaction Engine for textured objects was tested with the different RenderArrays with the camera path shown in Figure 3.22.

The triangle budget is rendered from the first nodes in the RenderArray whose sorted distance to the intersection point of the line of sight and the model is smallest. To avoid texture binding for each render node the texture binding of adjacent rendernodes in the RenderArray that have the same texture id of the parent volume are cached.

It can be seen from Figure 3.23, that a rendering performance above 14-15 frames per second can be achieved with texture mapping in any viewing condition with a RenderArray of 689 render nodes.

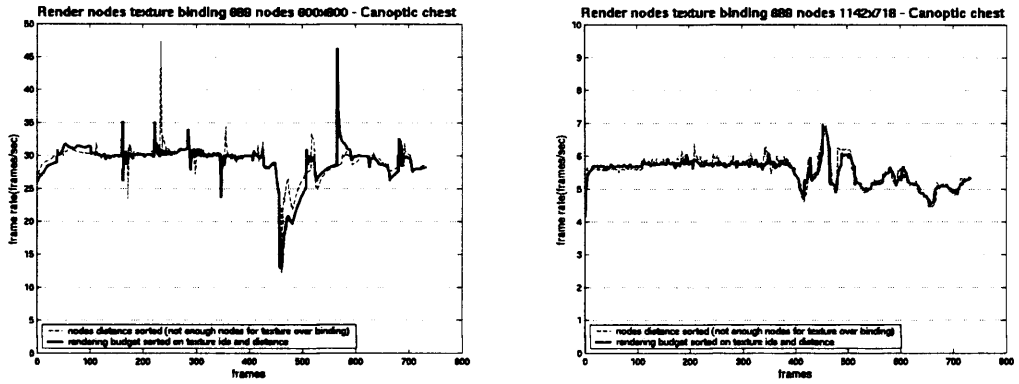


Figure 3.23: Triangle budget rendering performance using a fixed size RenderArray with graphics card support (left) and with CPU based rendering (right).

Whilst this works well for 689 render nodes, a problem arises when we wish to use a larger RenderArray of 7,525 render nodes (Figure 3.24). In this case there is a performance drop to almost 1 frame per second when using the graphics card. Specifically, we traced the problem to texture over binding, in the frames just after the camera's vertical accent (Figure 3.22). These frames are from the same fixed position with different orientations looking down and obliquely to the model, putting in view render nodes of different corner volumes. Even though several render nodes share the same texture id of the parent volume, render nodes of different textures are bound to have the same distance. This has the effect of alternating the position of their render nodes in the RenderArray and causing texture over binding. Our solution to this problem, is to sort the render nodes of the RenderArray according to distance as previously, but the number of render nodes used for the current rendering triangle budget chosen by the user are re-sorted according to texture id. During the sorting process, if render nodes have the same texture id they are sorted on distance.

Figure 3.24 demonstrates that our modified algorithm has solved the problem of texture over binding and met our performance aim of 15 frames per second in any viewing condition.

It can be seen in Figure 3.23 that the additional sort on texture id has negligeable performance impact.

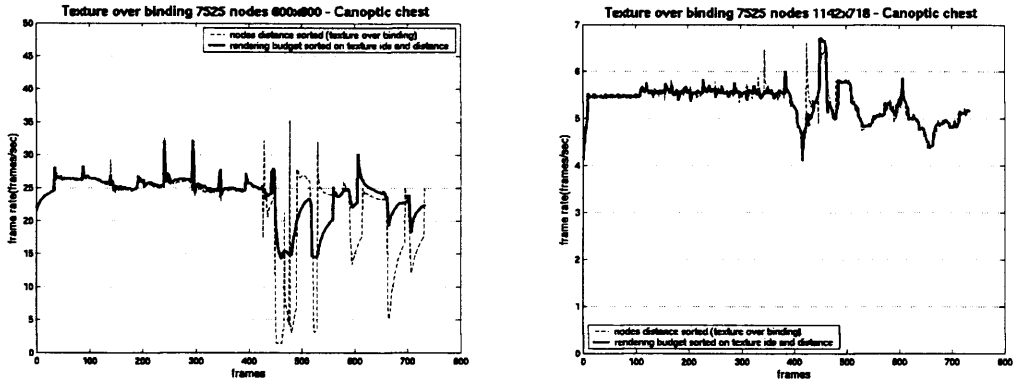


Figure 3.24: Triangle budget rendering performance using a fixed size RenderArray with graphics card support (left) and with CPU based rendering (right).

Figure 3.25 shows that shorter RenderArrays maintain the same performance characteristics.

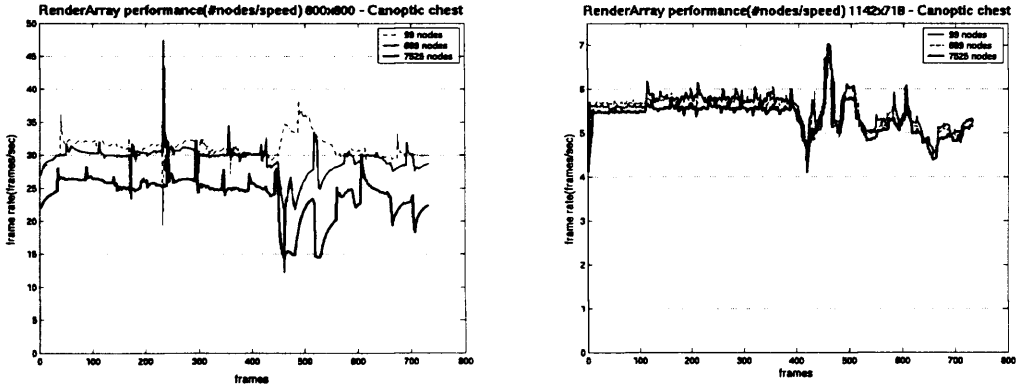


Figure 3.25: Triangle budget rendering performance using a fixed size RenderArray with graphics card support (left) and with CPU based rendering (right).

The Octree Interaction Engine texture extension was further tested with an outdoor laser scan data set of the facade of a Cathedral comprised originally of 44 textures, 1,160,186 triangles and 597,118 vertices. Figure 3.26 demonstrates the spatial accuracy trade-off when rendering a triangle budget of 4,000 from the same intersection point of the line of sight and the model with RenderArrays of 3,327 and 34,219 render nodes respectively. The Octree Interaction Engine automatically computed 667 relatively small new textures, and a 525 triangle

navigation skin. The textures were computed for volumes (3,327 nodes) one level higher than the leaf nodes (34,219 nodes).

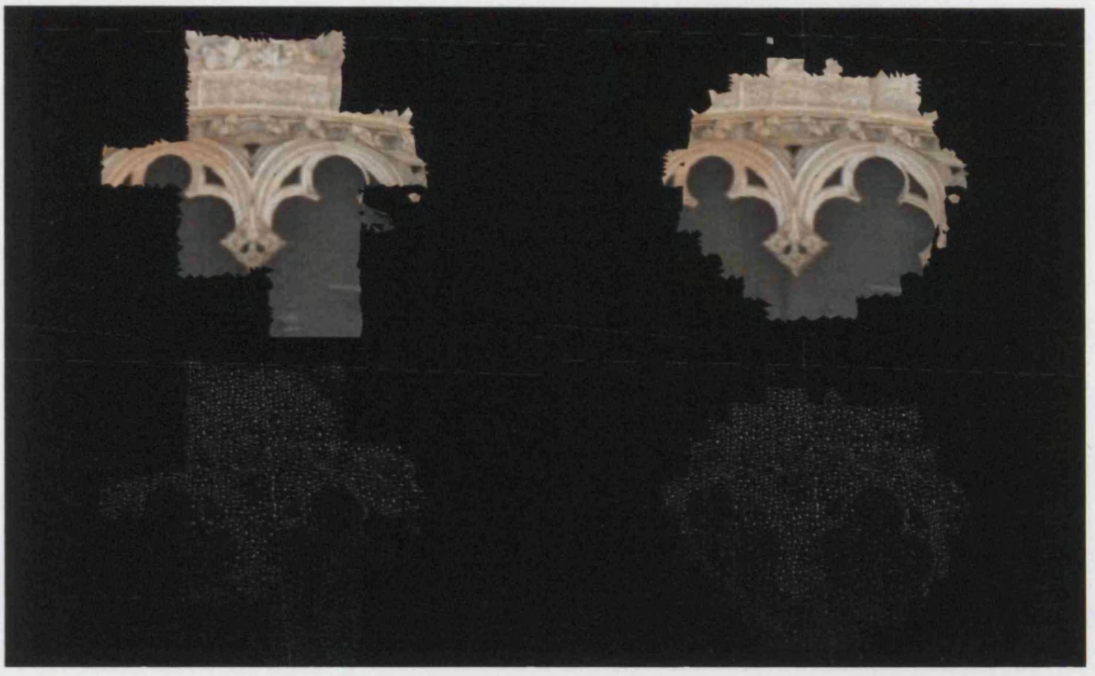


Figure 3.26: Octree Interaction Engine rendering of a scanned and textured type of object. Rendering budget of 4,000 triangles using RenderArrays of increasing size and spatial accuracy for the Batalha cathedral scan; *left:* 3,327 render nodes; *right:* 34,219 render nodes.

Figure 3.27 shows the same contents as Figure 3.26 but from afar. A larger rendering budget has been used in the third image from left, and a full triangle budget rendering used in the rightmost image. A long artefact can be seen in the rightmost image where the laser beam was at a very low angle of incidence to the floor, capturing spurious data nearby.

Figure 3.27: Progressive rendering of Batalha cathedral scan - from left to right: 4,000 triangle budget and RenderArray of 3,327 nodes; 4,000 triangle budget and RenderArray of 34,219 nodes; 38,803 triangle budget and RenderArray of 34,219 nodes; full 1,160,186 triangle budget.

The model was tested with the different RenderArrays with the camera path shown in

Figure 3.28.

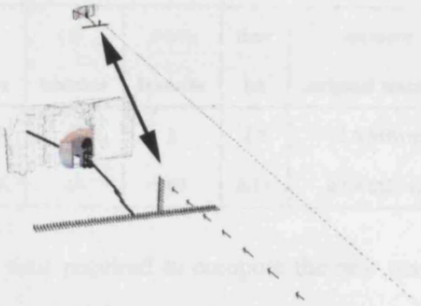


Figure 3.28: Camera path for the Batalha cathedral scan - *left:* zoom; *right:* increased distance view.

Figure 3.29 shows that a rendering performance above 14-15 frames per second was possible with a RenderArray of 3,327 nodes with the camera path of Figure 3.28

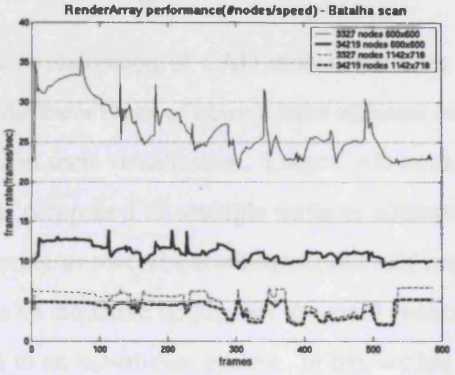


Figure 3.29: Triangle budget performance using a fixed size RenderArray with graphics card support (600x600) and with CPU based rendering (1142x718). It can be seen that with graphics card support and a suitable length RenderArray of 3,327 nodes it is possible to maintain frame rates above 20 frames per second.

Table 3.3 shows memory and performance results for the Octree Interaction Engine's extension for texture models. It demonstrates that, for the two examples used in this thesis, the newly computed textures require almost half the memory of the original textures that they replace. The memory saving is achieved because any original texture information that is not mapped to a model triangle is discarded. This is particularly useful as the graphics card used in our experiments has a maximum of 8 MBytes of video memory, thus the optimal capabilities of the card can be exploited with the smaller model. We note that our algorithm for computing

new textures requires less than 10 seconds to process textures for each models.

Model name	# triangles	# textures	#new textures	time (s)	memory (original textures)	memory (new textures)	memory (total model)
Canoptic chest	284,399	5	8	3.2	12.5 MBytes	7.2 MBytes	110 MBytes
Batalha -entrance-	1,160,186	44	667	8.13	47.9 MBytes	24.2 MBytes	312 MBytes

Table 3.3: Memory and time required to compute the new textures for the Octree interaction engine extension for texture models.

In Chapter 7 we outline our plans for extending the Octree Interaction Engine paradigm to multi-projector, parallel rendering of out-of-core models.

3.4 Octree Interaction Engine for CAD and volume objects

This section addresses the visualization of CAD models with the Octree Interaction Engine. As mentioned previously different types of objects have different intrinsic modeling properties that have to be considered in their visualization. Large CAD models comprised of thousands of objects, like the models comprised of multiple surfaces addressed in Section 3.1.5, do not benefit from surface clustering as several disturbing surface self intersections are created. Discrete hiearchical LoDs are an attractive option, but the extra memory required could force the visualization of the model to an out-of-core system. In this section we present some solutions to the problem.

The model used to study CAD models in this section was the power plant model used in University of North Carolina's Walkthrough project. The model comprises 1,185 objects, 12,748,510 triangles and 11,070,509 vertices. As with other large models presented previously in this chapter, the complete model could be loaded into less than 1 GByte of main memory, including colour attributes, with our normal and colour attribute compression algorithm PNORMS which is presented in the next section.

In common with [HDG96, CGG+04] we attempted with some success wireframe rendering of the bounding boxes of the hierarchical model. We used our depth buffer strategy presented in the previous section to ensure that the fine geometry would always be rendered over the wireframe boxes. Figure 3.30 shows the model being rendered with a triangle rendering budget of 131,485 triangles.

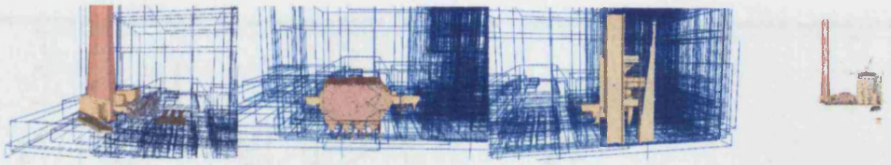


Figure 3.30: Hierarchical box rendering of UNC power plant model with a rendering budget of 131,485 triangles and 1,185 hierarchical boxes.

The camera path in Figure 3.31 was used to assess the performance of different rendering strategies with this model.

Figure 3.31: Camera path for the UNC power plant model starting from the right and moving in a clockwise direction with respect to the model.

One problem that becomes apparent with hierarchical box rendering, is the inherent granularity of the hierarchy. Even though the power plant is comprised of over 1,000 objects, objects can be nested within an object definition. Whilst nesting means less boxes to render, the boxes provide a crude idea of the underlying object. A second problem with hierarchical box rendering in the context of the Octree Interaction Engine, is that all the boxes are rendered in every frame (Figure 3.32). Such an approach does not accomplish our rendering performance aim of 14-15 frames per second.

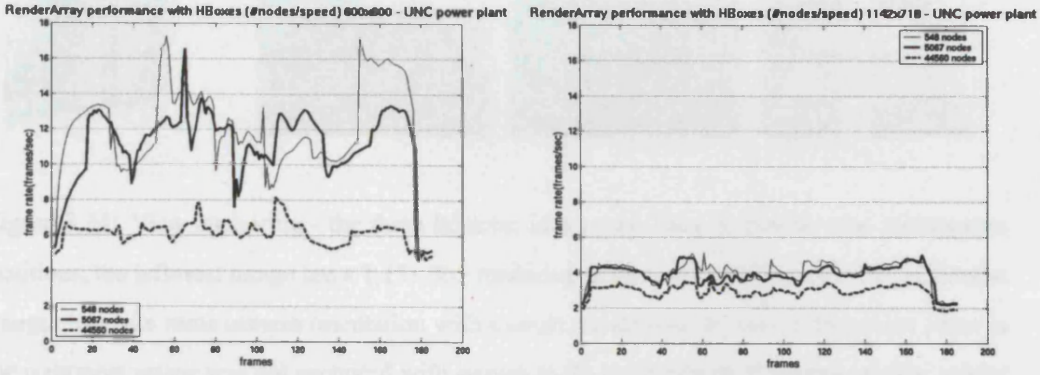


Figure 3.32: Rendering performance results using RenderArrays of increasing size and spatial accuracy with a fixed 4,000 triangle budget and hierarchical box (HBoxes) rendering for the UNC power plant model.

The performance limitations mentioned earlier are likely resolved with a more powerful graphics card. One interesting property of the Octree Interaction Engine mentioned earlier, is ***implicit*** occlusion culling, this property is illustrated in Figure 3.33, where the rendering of the full model differs very little from the set rendering budget of 258,970 triangles.

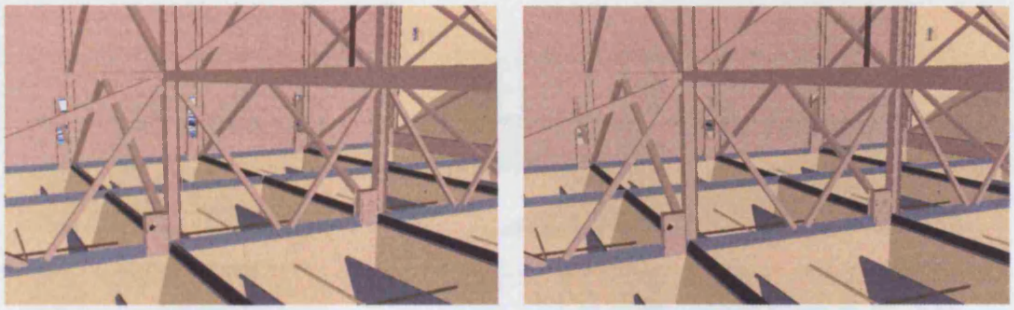


Figure 3.33: Near full occlusion in UNC power plant model - *left:* rendering budget of 258,970 triangles *right:* full rendering budget of 12,748,510 triangles

A useful extension when viewing the powerplant model is ***view anchoring***, where the user can freeze and re-use the intersection point of the line of sight with the model, whilst inspecting the same rendered triangles from other angles. In Figure 3.34, the third and fourth leftmost subimage differ only in a small translation or strafe to the right with large rendering differences. View anchoring on one of the pillars of the cage like object enables the bird's eye type of view illustrated in the second leftmost subimage.

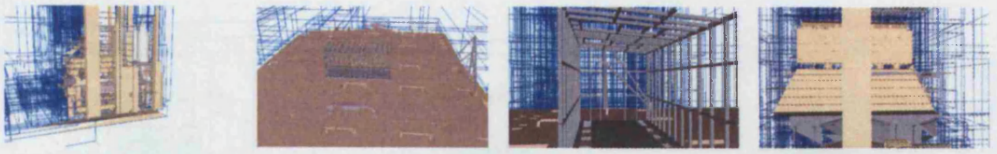


Figure 3.34: View anchoring - the three leftmost images are from anchored view intersection positions, the leftmost image has a 1,151,365 rendering budget; the camera in the two rightmost images have the same camera orientation with a small translation; the view intersection point in the rightmost image was not anchored with respect to the camera in third leftmost image, whilst the second leftmost subimage was.

The idea of view anchoring can be taken further through the introduction of the idea of quicktime VR like hotspots. A user could browse the model by clicking on distant floating icons in parts of specific interest for the task. Here the difference is that the user or creator of the object would annotate in the hotspot a specific camera position, possibly camera orientation(s) and a frozen intersection point of the line of sight with the model near the feature of interest. Figure 3.35 shows three such hotspots.

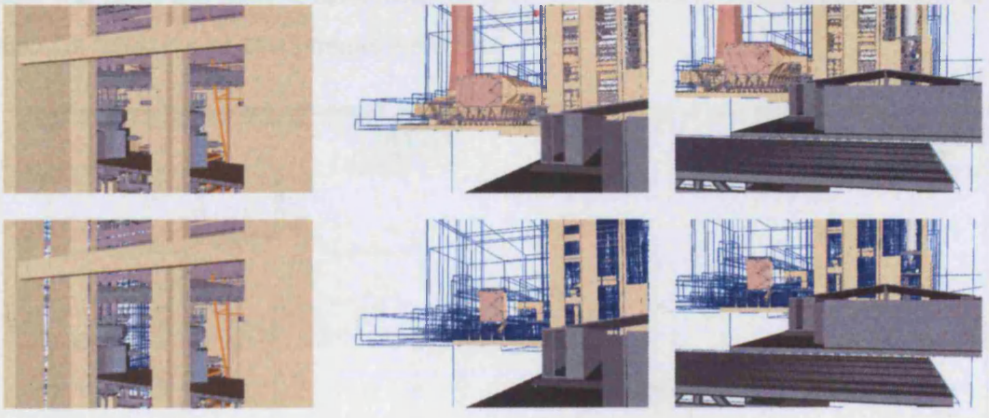


Figure 3.35: Hot spot viewing with hierarchical box rendering - *bottom:* 131,485 triangle budget; *top:* full model rendering.

As with multiple surface rendering, user controlled octree depth rendering was evaluated, Figure 3.36.

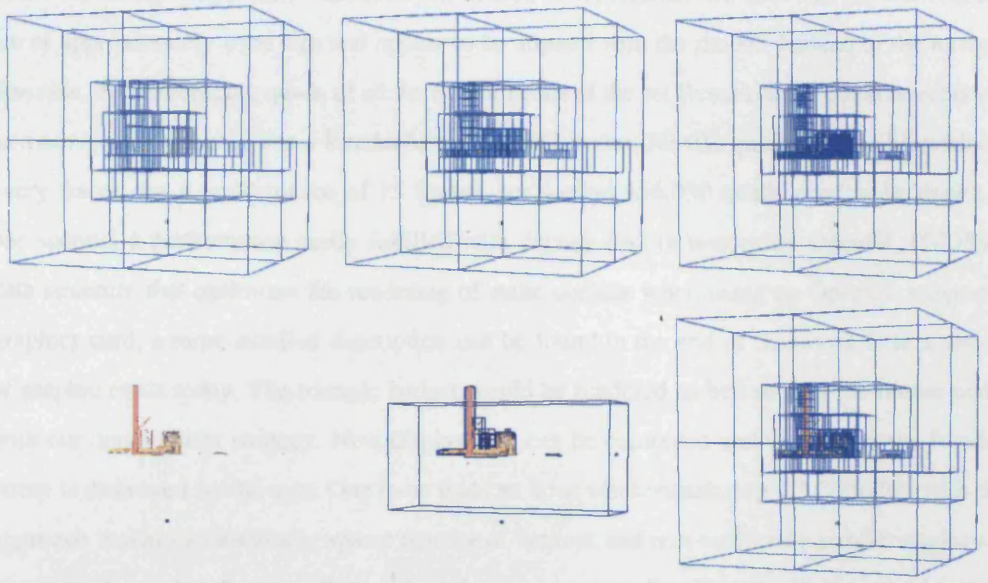


Figure 3.36: Octree depth rendering of the UNC power plant model - *top:* from left to right: depth 3, 4 and 5; *bottom:* from left to right: full model (12,748,510 triangles); full model and 1,185 hiearchical boxes; full model with octree depth 5 rendering.

Figure 3.37 shows that octree rendering of depths 4 and 5 with a RenderArray of 5,067 nodes was faster and more scalable than hierachical box rendering and reached our aim of a minimum frame rate of 14-15 frames per second.

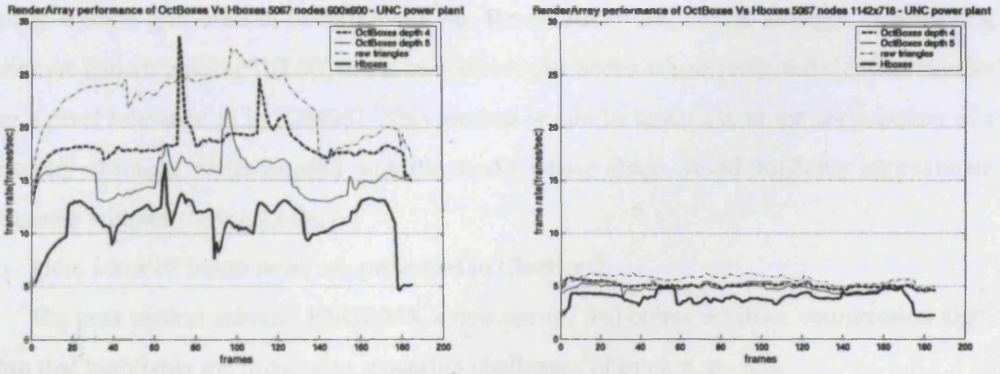


Figure 3.37: Rendering performance comparison for the UNC power plant model of: hierarchical box rendering (Hboxes) with a 4,000 triangle budget; octree depth level rendering and a 4,000 triangle budget; rendering of 4,000 triangle budget (raw triangles).

An alternative to octree depth rendering, is illustrated in Figure 3.2 where the render nodes

of the rendering budget are rendered in full instead of wireframe. We note that the rendernodes are of approximately even size and appear to be aligned with the model. Instead of the *navigation skin*, the front facing quads of all the render nodes of the set RenderArray could be rendered as a *navigation volume*. For a RenderArray of 5,067 nodes, 30,402 quads need to be rendered every frame, for a performance of 15 frames per second 456,030 quads need to be drawn in one second, a performance easily fulfilled with *display lists* (a temporary OpenGL [WDS99] data structure that optimizes the rendering of static content when using an OpenGL supported graphics card, a more detailed description can be found in the end of Section 2.1) in a variety of graphic cards today. The triangle budget would be rendered as before over the render nodes with our depth buffer strategy. New display lists can be computed and used when the RenderArray is destroyed by the user. One issue that can arise when visualizing CAD models with this approach is that the inherently sparse density of vertices and non-uniformly sized triangles can create render nodes of very different sizes that are not spatially aligned with the sub objects of the model and hence might not provide a good approximation of all of the geometry well. Figure 3.38 shows the rendering result of a part of the UNC power plant model in the case where render nodes are aligned with the model.

The main objective of this thesis was to address non-uniform model reduction. This chapter has given a solution for the interaction and triangle selection of models in general and included a wireframe octree depth rendering suited to CAD models.

One approach that could be used to further explore visualisation of CAD models and counter the scarcity of vertices and triangles within such models, would be to use the hiearchical regular volume grid used in [GM05] with our RenderArray and Octree strategy. A rendering technique called splatting [RL00] could be used to splat nodes whose projected size was smaller than a pixel tolerance as in [GM05]. This method would be amenable to the construction of a hierarchy of render nodes aligned with the model whose shape would implicitly approximate the model without storing LoDs.

More ideas of future work are presented in Chapter 7.

The next section presents PNORMS, a new normal and colour attribute compression algorithm that highlights the increasing rendering challenges of in-core models.

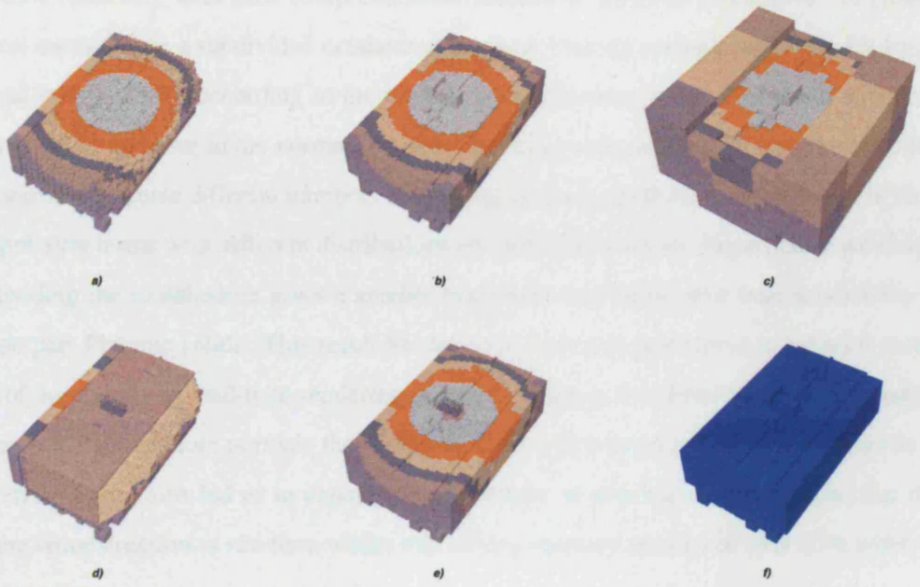


Figure 3.38: Rendering render node volumes and triangle budget rendering of original triangles - a) 54,204 nodes and zero triangle budget; d) 54,204 nodes and 248,432 triangle budget; b) 6,782 nodes and zero triangle budget; e) 6,782 nodes and 4,000 triangle budget; c) 677 nodes and zero triangle budget; f) 677 nodes approximating the volume of the full model rendered in d).

3.5 PNORMS-Platonic Derived Normals for Error Bound Compression

3D models of millions of triangles invariably repeatedly use the same 12-byte unit normals. Several bit-wise compression algorithms have been developed for efficient storage and progressive transmission and visualization of normal vectors. However such methods often incur a reconstruction time penalty which, in the absence of dedicated hardware acceleration, make real-time rendering with such compression/reconstruction methods prohibitive. In particular, several methods use a subdivided octahedron to create look-up normals where the bit length of normal indices varies according to the number of subdivisions used. Not much attention has been given to the error in the normals encoded by using such schemes. We show that different Platonic solids create different numbers of normals for each subdivision or bit length in bit-wise compression terms with different distributions and associated errors. In particular we show that subdividing the icosahedron gives a smaller maximum and mean error than subdivision of its counterpart Platonic solids. This result has led us to create an alternative to bit-wise compression of normal ids for real-time rendering in which we use a five times subdivided icosahedron to create 2.5 times more normals than obtained from a five times subdivided octahedron, with less error. It has also led us to exploit the advantages of absolute normal indices that do not require reconstruction at run-time whilst still having memory savings of over 83% when using 2-byte indices.

We present results using 2-byte indices for a target maximum error of 1.3° degrees and 4-byte indices for a maximum error of <0.1°. We present two hierarchical encoding methods. A fast method which allows one dynamically to encode large sets of modified triangles useful for example in the context of a VR user BSpline/object modelling task, and a slower but more accurate method that caters for symmetry present in the subdivision solid being used. In addition, different levels of a database allow for different cartoon like shading effects at run-time. The advantages of these databases are that they can be re-used for any object and have bounds on the maximum errors of normals encoded for any geometry known or unknown such as when new objects are added to a scene. This error bound is also independent of the size and distribution of the normals of the object that we wish to model. In order to visualize the distribution of the errors in the normals of large models a simple 1-byte color encoding algorithm was developed (right most sub image of Figure 3.39).

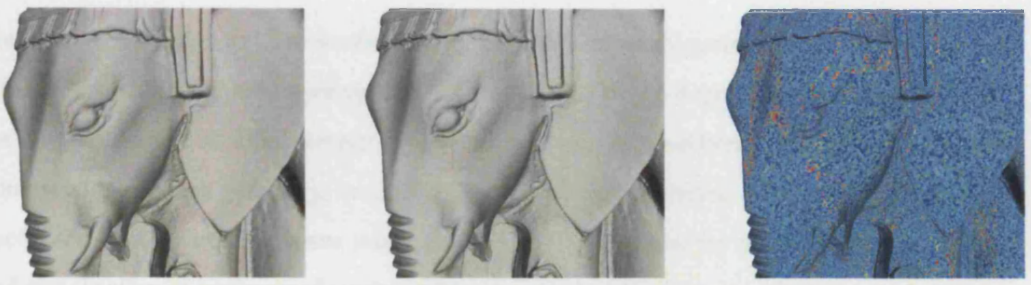


Figure 3.39: *Left:* Flat-shaded rendering of the original 12-byte, 10 million triangle normals; *centre:* with a max error of 2.5 degrees using 27,300 12-byte normals from a 2-byte index icosahedron database (encoding in 95secs) *right:* colour coded error distribution, maximum error in red.

3.5.1 Introduction

Many applications such as global models of fluid flow in meteorology and oceanography employ spherical geodesic grids that are based on a subdivided icosahedron [RRH+02] which leads to a quasi-uniform distribution of points without singularities at the poles [G.90]. Each triangle of an icosahedron, octahedron or tetrahedron (Figure 3.40) can be subdivided to create 4 new sub-triangles by inserting 3 new vertices at the midpoints of a triangle's edges [WDS99]. Several other methods of subdividing triangles have been studied in geographic information systems (GIS) [G.90], although quadrisecting [GVSS00, vRLJHK05, TR98] or hexasecting of octants [Dee95] at midpoints have proven to be popular.

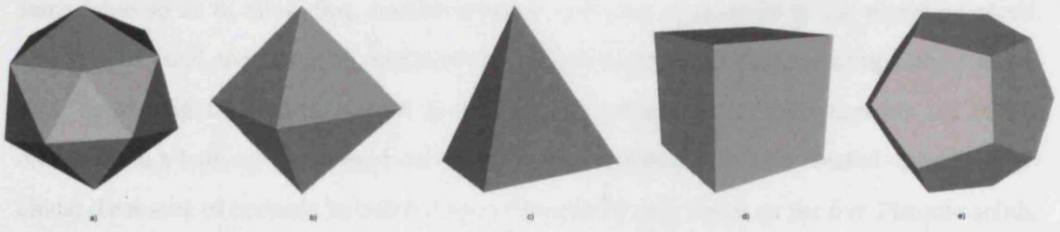


Figure 3.40: Base Platonic solids; a) icosahedron b) octahedron c) tetrahedron d) cube e) dodecahedron

The vectors representing these midpoints can then be normalized to unit length and hence projected onto a sphere (Figure 3.41). The normals of these sub-triangles constitute a finite set of normals with different distribution characteristics for each solid. Normals within these sets can be re-used by several triangles or vertices of an object for both flat and Gouraud shading. We study these distributions in the context of using look-up databases for surface normal com-

pression. The method to be presented allows for 83.4% memory savings for storage of normal attributes, with a bounded imperceptible maximum error of 2.5 degrees without a requirement for any reconstruction time. Several compression strategies have been developed for efficient compression of mesh geometry, connectivity/topology and attributes. These methods are very useful for saving hard disk space when storing large meshes and for progressive transmission and visualisation of compressed meshes over a limited network. However these methods require computation time for reconstruction in order to retrieve and decompress the original data which might be run-length encoded or *gzipped* along with other, different compressed attributes such as geometry, connectivity and colour, making the decompression process in a real-time setting prohibitive in the absence of dedicated hardware support.

Our approach may be regarded as following an analogy with GIF[3]image compression rather than JPEG. The GIF image format sets the colour index to a single byte to access 256 colours in a table rather than utilizing a variable bit length, for example, to exploit patterns which could be run-length compressed. The latter approaches tend to lead to somewhat involved arithmetic coding and decoding procedures which can prohibit real-time reconstruction. Viewers of animated GIFs, unless they are used within a web-browser, are able to render and switch images at a very fast rate by look-up of colour indices in a table for the colour of each pixel. In contrast, JPEG images are able to provide a fast overview of the full image but require more time to switch between two full resolution images. Similarly, from the observation that meshes of high resolution scanned data repeatedly use the same, or almost the same, unit normals we build look-up databases of a finite set of normals. Unlike GIF however, our tables contain coarse normals of a base solid along with finer normals of deeper subdivisions in the same table so as to allow fast, multi-resolution querying of normals in our encoding phase. Once the original, true normals computed from the vertices of the triangles comprising a model have been assigned a normal index in the encoding phase then 12 byte normals are simply obtained via a look-up at run-time, for example for rendering. We have created several hierarchical databases of normals by subdivision of triangular nets based on the five Platonic solids. We created databases for 2 byte normal indices (2^{16} = 65,536 possible lookup normals in the table) and 4 byte (2^{32} = 4,294,967,295 normals). Different Platonic solids produce different numbers of triangles at each level when we repeatedly quadruple the number of their initial triangles. In Table 3.4, the row labelled zero on the left shows the number of initial triangles of each of the base Platonic solids whilst subsequent rows show the number obtained on subdivision. This section shows which of the five solids yields the results in terms of which one

[3]graphics interchange format

achieves the smallest representation error. This is given a restriction on the number of normals each solid can generate so as to not overflow the number of possible normals accessible with 2 bytes or with 4 bytes. Whilst our method creates absolute normal indices that do not require decoding, it is still based on creating different subdivisions of Platonic solids. Since the number of subdivisions affects the bit-length of bit-wise compression methods, the errors studied and presented in here are directly applicably to such methods. Namely we consider the number of subdivisions, the number of available normals, symmetry between normals from adjacent base triangles and more importantly the error versus memory cost.

We briefly review related work in Section 3.5.2. In Section 3.5.3 we study various subdivisions. In Section 3.5.4 we present a fast and a slower, but more accurate, method of encoding a normal into a database with a bounded maximum error of 2.5 degrees using a five times subdivided icosahedron with 2 byte normal ids rather than the conventional 12 bytes. We found this accuracy suitable for visualization of 3D models. The fast method carried out without dedicated hardware is quick enough using a G4-500Mhz PowerPC CPU to allow dynamic re-encoding of large sets of modified triangles for example, in a VR modelling task. The slower but more accurate method is for applications that require sub-degree accuracy. We note that the encodings produced by both methods can be accessed at the same speed at run-time. In Section 3.5.5 we present results. We compare timings and the maximum errors of both the fast and slow encoding methods applied to the same models which range from a whole scene such as the UNC power plant CAD model to scanned statues whose sizes vary by several orders of magnitude. Table 3.5 also shows that the error is bounded to 2.5 degs and 1.3 degs respectively with any model, whilst the timings scale linearly with the size of the input model. A simple colour encoding algorithm is presented in Section 3.5.5.4. In Section 3.5.6 we discuss a fair comparison of the distributions of the normals derived from the different Platonic solids and draw conclusions in Section 3.5.5. We conclude the chapter in Section 3.6.

3.5.2 Related compression work

Compression of mesh geometry [Cho97], connectivity [TR98], and attributes [BPZ99] is a mature field. In this section we review methods that specifically compress normal attributes.

Creating a database of normals for look-up is not new [BWK02, Dee95, RL00, THRL98]. For example, Deering [Dee95] uses the GPU to decompress and look-up normals from a database using 18 bits for each triangle normal index. 3 bits are used to encode the +/- sign of a normal's x, y, z components, hence determining the octant in which a normal lies, 3 bits determine the sextant, and then two 6 bit numbers encode the phi and theta spherical coordinates of a normal (these 18 bits normal indices are then delta encoded). In contrast, compression

methods based on octahedron subdivision [BWK02, THRL98] use 3 bits (to locate in which of octants 0-7 the normal lies) and several 2 bit pairs where each pair represents one subdivision (the 2 bits locate one of the 4 subdivided triangles). Each 2 bits requires decoding to evaluate the full normal of a triangle (which can use 13 bits [BWK02] for a five times subdivided octahedron or 5-8bits when gzipped), hence the number of subdivisions affects the bit length and number of computations. Not much attention has been given to the error of the encoded normals which is also related to the number of subdivisions. Since the number of subdivisions affects computation in decoding, memory usage of a model, and error, we study and show that different Platonic solids generate very different numbers of normals for the same subdivision level or bit pair and have different normal errors and distributions associated with them. These studies have shown that enough quasi-uniformly distributed normals are created by 5 subdivisions of an icosahedron to adopt absolute normal indices, whilst only using 16 bit normal indices that do not require reconstruction. Specifically a five times subdivided icosahedron generates 2.5 times more normals (20,480) than an octahedron subdivided five times (8,192 normals), allowing for a maximum error bound of 2.5 degrees and 1.3 degrees using our two new hierarchical encoding schemes respectively with any model (Table 3.5). The size of the single database that stores the normals for the results shown in Figure 3.39 is ~328k (27,300x12bytes) which we regard as negligible (see Table 3.6) for most devices. We note that such error bounds are possible because we sample the whole space of possible normals, rather than optimizing the use of normals for a single object. This allows a small database to be used for several objects without having to create new databases.

Taubin et al. [THRL98] also parameterize a sphere using a method based on subdivision of an octahedron. They subdivide each base triangle of the octahedron into four and compress the normal id associated with the sub-triangles. Such bit-wise compression imposes a reconstruction time which, without dedicated hardware, can be prohibitive for real-time rendering of large meshes. Our method stores the normals of all subdivision levels so as to allow for hierarchical querying of normals in the encoding phase. We note that methods that use a subdivided octahedron do not address the symmetry between adjacent triangles of the base solid. Consequently, when a normal being encoded lies almost halfway between two adjacent base triangle normals, an incorrect base triangle normal can be chosen, as the inverse cosine of a dot product blindly gives an angle that can be deemed to be the smaller from either side. This results in a less than optimal representative normal being found. Symmetry is also present between adjacent triangle normals at deeper levels of the subdivision, across the sphere and at several locations. We present an encoding method based on subdivision of Platonic solids that

caters for symmetry (Section 3.5.4.2) and we show how the encoding error is reduced. Once the true, computed normals of a model are assigned to a representative normal, no reconstruction is necessary. To compare our two encoding methods, we carry out an error analysis using the computed, true normals obtained directly from the geometry. Whilst an octahedron is easy to generate in comparison with the icosahedron, we found as in the GIS work [G.90, RRH+02] in the past 60 years, that the icosaherdon yields a more uniform distribution of points which, in our context, translates to smaller maximum and mean errors when used in conjunction with our encoding. Our databases of representative normals derived from the icosahedron have been made available online.

One goal that is common in normal attribute compression methods is that of searching/finding rapidly a representative normal in the collection of created point samples that best approximates the normal being encoded. Von Rymon-Lipsinki et al. [vRLJHK05] use a binary search with the GPU to find a representative normal. Tables of errors for point sampling of functions that parameterize a sphere have been computed [Slo81]. We note that the normals in our databases were created through a less deterministic method by altering geometrically a solid rather than via a parameterization. This strategy has allowed us to create a fast hierarchical, multi-resolution querying system for finding a representative normal in a database corresponding to a given normal using exclusively dot-products without spherical coordinate conversions [Dee95] or graphics hardware [vRLJHK05].

Deok-So et al. [KCK04] note that, in previous work, normal ids are created independently of the concentration of normals in a model. They also note that a triangle normal is bound to be referenced in neighbouring triangles and thus use normal clustering to find representative normals. Furthermore, they use a relative indexing scheme to further compress the sequence of normals. It would be desirable to encode on the fly large sets of new triangle normals when, for example, one modifies a mesh. However it is not clear that such high compression methods can be used in such a run-time context. Deok-So et al. also note that, whilst several excellent compression methods developed in the past decade achieve excellent compression rates, not much attention has been given to the errors incurred in the compression. We refer the reader to a more extensive review in [KCK04].

Guskove et al. [GVSS00] parameterize a subdivision surface using a method in which multiple levels of detail can be obtained from a base, coarse mesh and a corresponding offset value. Each vertex can be stored by means of a single float. This representation requires the mesh to be semi-regular, which in some cases requires the object to be re-meshed and is not suitable for non-manifold or CAD objects. This problem is often shared also by approaches that look for

sequences of triangle strips [BPZ99]. Some methods such as [Cho97] store a compressed representation in main memory and use a fast decoder to render in real-time. It is not clear whether this approach allows for encoding at run-time of large triangle sets. Our method of encoding triangle normals to a single database for the scene does not require triangle strips and does not require a surface to be manifold, as each triangle normal is considered/encoded individually (see Table 3.5 for results with both hierarchical and scanned objects).

3.5.3 Platonic normals

Our compression method consists of three different phases. The first phase which is only performed once consists of the subdivision of Platonic solids and construction of hierarchical databases that once created and saved can be used for any object, with bounds on the maximum error. The second phase is an encoding phase, which consists of finding a representative normal in the database for each true normal computed from the object or scene geometry. Once a representative normal is found, an id denoting the position of the representative normal in the database is stored. These normal ids, along with the database, can be readily saved to file. The third and final phase involves a look-up of the full 12 byte normals for rendering at run-time using the normal ids. We show in Section 3.5.5.2 that there is no penalty in this look-up.

All five Platonic solids: the icosahedron, octahedron, tetrahedron, cube, and dodecahedron (Figure 3.40), share the following interesting properties. The vertices of each solid all lie on a sphere; each solid has the same dihedral angle between its adjacent polygons (138.190°, 109.471°, 70.529°, 90°, 116.565° respectively); their polygons are regular; all their vertices are surrounded by the same number of faces; and the solid angle subtended by each face is identical [WEI06]. We tried basing our subdivision on other triangle mesh objects that did not share these properties, such as an LoD of the Stanford bunny, but found the resulting distributions to sample disproportionately different areas of the unit sphere. This rendered them unusable. For example, the property that all vertices lie on a sphere, allows one to divide a sphere into manageable equal size parts. This in turn helps the subdivision (Figure 3.41) or construction phase of building the hierarchical databases of representative normals described in Section 3.5.3.1. The property of a Platonic solid having the same dihedral angle amongst all adjacent polygons helps the encoding phase of the accurate approach presented in Section 3.5.4.2. We mention polygons, rather than triangles, because the dodecahedron has pentagons and the cube has square faces. In Section 3.5.3, we have triangulated the squares and pentagons into different triangle arrangements or polygon nets so as to be able to subdivide and study the resulting distributions of normals.

3.5.3.1 Subdivision: Hiearchical database contruction of representative normals

In this section we describe how to create hierarchical databases of representative normals. We note that contrary to other methods, we store the normals of every subdivision level. This strategy allows one to make fast hierarchical queries of the database at run-time for encoding new true normals on the fly. If we insert and normalize new vertices at edge midpoints for each base triangle as illustrated for triangle ABC in Figure 3.41, we obtain four sub triangles which can be used to compute four finer grain representative normals for the sphere projected area covered by the base triangle.

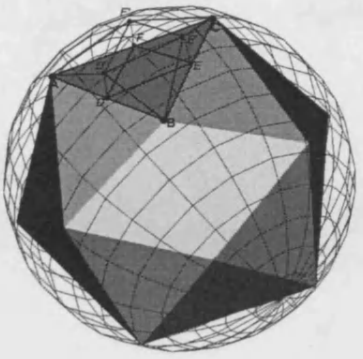


Figure 3.41: Subdivision of the base triangle ABC of the icosahedron, into triangles ADF, DBE, ECF, and FDE; inserted vertices at the midpoint of edges AB, BC, CA are normalized/projected to a unit sphere to form triangles AD'F', D'BE', E'CF', and F'D'E'.

Each Platonic solid is formed by a different number of base polygons; 20, 8, 4, 6 and 12 respectively, for the isosahedron, octahedron, tetrahedron, cube, and dodecahedron. Before we can subdivide the cube and dodecahedron we need to create triangle tessellations of their square and pentagonal faces. These triangle tessellations or polygon nets define arrangements of triangles that can affect the resulting normal distributions. The nets we studied are shown in Figure 3.42. After tessellation, the cube consists of 12 base triangles and the dodecahedron of 36 base triangles.

Subdividing a Platonic solid at run-time would require connectivity information for the Platonic solid, geometric updates to it, and memory management, and hence run-time overheads. We opt for creating these object-independent subdivided models once, offline, and store only the generated representative normals in an orderly fashion in an array. This is similar to a GIF file's colour palette/array that is saved in the header file of a gif image. Unlike GIF however, we store the hierarchical databases separately from an object in order to be able to re-use the same database for multiple objects. The first 20 base normals of an icosahedron occupy the

first 20 positions of the array, the four subdivided normals of the first base triangle are added to the array next, and occupy positions 20, 21, 22, 23. The four subdivided normals of the second base triangle, are stored next in locations 24, 25, 26, 27. Further levels of subdivision are added in the same manner to the array.

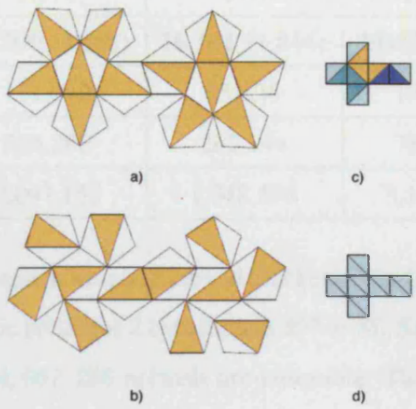


Figure 3.42: Polygonal nets a) dodecahedron b) dodecahedron c) cube d) cube

After subdividing each solid five times (Figure 3.43) one can observe how uniform the distribution of normals obtained is by looking at how uniform the resulting triangles are in size and how regular the tessellation is. The number of generated representative triangle normals may or may not contribute to a reduction in orientation errors of an encoding as we will show in Section 3.5.5. We note that the five times subdivided icosahedron in Figure 3.43 generates 20,480 representative normals (first column of Table 3.4, row 6 - the first row has the number of base triangles and is not a subdivision), whilst the five times subdivided octahedron generates significantly fewer normals (8192, second column of Table 3.4, row 6). However we can see from Figure 3.44 that the 4 times subdivided icosahedron with 5,120 representative normals is more regular than the 5 times subdivided octahedron. Figure 3.43 and 3.44 also show that the different polygonal nets (shown in Figure 3.42) for the cube and dodecahedron did not significantly affect the overall characteristics of the distributions of their nomals. The tetrahedron appears to generate the least regular distribution.

	icosahedron	*octahedron*	*tetrahedron*	*cube*	*dodecahedron*
0	20	8	4	12	36
1	80	32	16	48	144
2	320	128	64	192	576
3	1,280	512	256	768	2,304
4	5,120	2,048	1,024	3,072	9,216
5	**20,480**(27,300)	8,192	4,096	12,288	**36,864**(49,140)
6	81,920	**32,768**(43,688)	**16,384**(21,844)	**49152**(65,532)	147,456
7	327,680	131,072	65,536	196,608	589,824
8	1,310,720	524,288	262,144	786,432	2,359,296
9	5,242,880	2,097,152	1,048,576	3,145,728	9,437,184

Table 3.4: Maximum indexable normals in curved brackets including all normals of each subdivision level for each Platonic solid, for 2 byte indices $2^{16} = 65,536$ normals are indexable. For 4 byte indices $2^{32} = 4,294,967,295$ normals are indexable. The number of triangle normals of the last subdivision level for 2 bytes is highlighted in bold.

Depending on the application requirements, a developer can use databases of representative normals constructed with 2 byte or 4 byte indices. If we use 2 bytes for each triangle normal index instead of the 12 bytes of the original normals, the statue of Lucy with 28 million triangles takes 56 Mbytes of memory instead of 336 Mbytes. Storing the five levels of a database derived from the subdivided icosahedon is not expensive (20+80+320+1,280+20,480=27,300x12bytes=327 Kbytes), but has to be taken into account in the total number of accessible normals for 2 bytes shown in brackets for each solid in Table 3.4. Overall, an 83.4% memory reduction can be achieved. We address the encoding error in Section 3.5.5.

An icosahedron can only be subdivided five times before the number of normals that can be indexed by means of 2 bytes is exceeded: 20 (the number of base normals)+80+320+1,280+5,120+20,480 = 27,300 < 65,536 normals. A sixth subdivision, shown in the first column row 7 of Table 3.4, would create 81,920 which would exceed the 65,536, 2-byte budget. It is notable that subdivision of a cube leads to a number of triangle normals (12+48+192+768+3,072+12,288+49,152=65,532 < 65,536) very close to the theoretical limit of accessible normals for 2-byte indices. In this case, very little address space is wasted but unfortunately having more representative normals does not translate directly into a lower encoding

error as we shall see in Section 3.5.5.

In an attempt to maximize the address space of 2 byte indices, we created a master object of normals that included two hiearchical databases obtained from 5 subdivisions of icosahedra that had been rotated a little to create an offset with respect to each other [27,300+27,300=54,600]<65,532. Each of the two databases obtained from the master object had an offset to their starting position in the array. A query would then search both databases for the best representative normal for a given true normal and would return the corresponding normal id. We tried different rotation offsets and noted that whilst the mean error decreased by having more representative normals available the maximum error did not. This is owing to the fact that the location of the second database will align with the first database in some places and hence not contribute to reducing the maximum error in those places. The nets that we built for triangulating the square and pentagonal faces of the cube and dodecahedron were constructed with two criteria in mind: the encouragement of local uniformity of the meshes containing the largest numbers of vertices possible (Figure 3.42; 6a and 6c), and a global uniform criterion (Figure 3.42; 6b and 6d). As can be seen in Figure 3.43 and 3.44 ((d) and (e)) these tessellations unfortunately did not improve the overall characteristics of the distribution of the representative normals obtained from the subdivided cube and dodecaherdron.

3.5.3.2 Selection of candidate Platonic solids

It can be seen from Figure 3.43 and 3.44 that the tetrahedron, cube, and dodecahedron lead to very non-uniform samplings of the sphere. This means that higher levels of subdivision which introduce more normals do not lead to normals being added uniformly over the sphere. Consequently, as one might expect the encoding accuracy of the representative normals derived from these solids is unlikely to improve in a linear manner as triangles are quadrisected. Furthermore, having what are effectively different resolutions in the same level of subdivision introduces a complication when designing tolerances for normal encoding at a given level. Our accurate encoding method works best for more uniform resolutions that require only a single tolerance that covers the whole subdivision level, as is the case for the octahedron and the icosahedron. Nevertheless, results are included for the first subdivision levels of the tetrahedron (Table 3.7) which start off as having uniform-like base triangles, and then degenerate to increased non-uniformity.

We used the triangulated cube and dodecahedron to explore different shading effects. However both these solids present a problem. They have base triangles which are on the same plane, unlike the other three solids. Consider for example the square face on the top of the cube which is tessellated into two triangles sharing an edge. Although when new vertices are introduced at mid-points of the edges and projected onto a sphere their different locations give rise to differ-

ent normals, there will be essentially duplicate normals in the base hierarchy for the cube and dodecahedron. Consequently any base normal from triangles on the same plane can be considered as the one to make the smallest angle error with the normal one is encoding, although only one of the base normals actually then yields subdivided normals that represent the normal more faithfully. However, our accurate method of encoding (Section 3.5.4.2) addresses symmetry by considering all normals in a given level of the database within a tolerance (so that, for example, both triangle normals from the top of the cube will be considered). We explored different tolerances for different levels of the cube and dodecahedron but found the databases derived from these solids to be inadequate for representing the space of all possible normals of a potential scene.

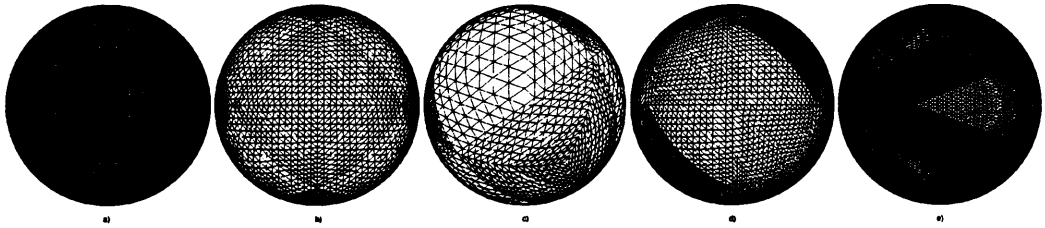


Figure 3.43: Platonic solids subdivided five times; a) icosahedron b) octahedron c) tetrahedron d) cube (using polygonal net 6c) e) dodecahedron (using polygonal net 6a).

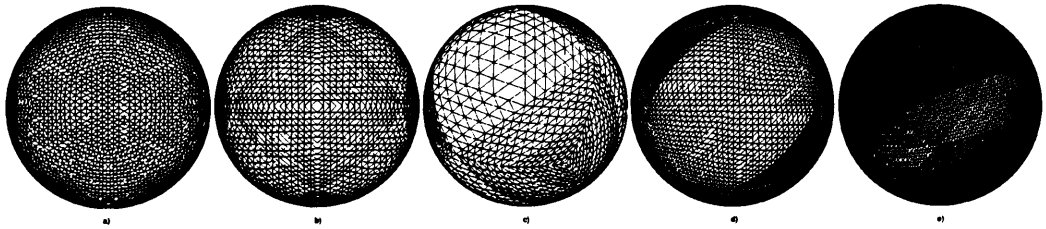


Figure 3.44: a) Icosahedron subdivided four times b) octahedron subdivided five times c) to e) solids subdivided five times c) tetrahedron d) cube (using polygonal net 6d) e) dodecahedron (using polygonal net 6b).

3.5.4 Encoding

Now that we have described how we created the hierarchical databases, we present two methods for encoding or finding a representative normal in the database for a given computed, true normal. We present a fast method suitable for encoding large sets of triangle normals at run-time and a slower but more accurate method, for applications that require more precision. We also note that the resulting orientation error in an object's normals not only depends on the spacing

of the created normals, but also on the encoding function that finds an adequate representative normal.

3.5.4.1 Fast encoding

For each triangle of a model, typically during reading the model from a file, we may readily compute its true normal with a cross product of its vertex pairs. We then compute the dot product of this 12 byte, true normal with all the normals of the base (unsubdivided) Platonic solid. Note that the normals in the database are full 12 byte normals and that the base normals occupy the first positions in the array database. The inverse cosine of this dot product will indicate which of the base normals has the smallest orientation error relative to the orientation of the true normal and hence approximates it better. If there are normals at a higher level of subdivision in the database, we further compare the true normal with the 4 finer-grain normals of the base triangle normal that had smallest error in the first comparison. In order to assist the querying process we pre-compute the start and end offsets for the beginning and end of each level in the database. For example, the start offset for the 5th level of subdivision of the icosahedron is the sum of the number of base triangle normals plus the number of all normals generated from the previous levels. The advantage of storing these start offsets is that we can use a simple relative indexing to access the four normals in which we are interested at each level simply by knowing the level, the start offset for that level, the relative number/or position of the triangle in the previous level and the quadrisected triangle we are examining (Equation 3.1).

$$\begin{aligned} normalid = startoffset + ((previousLevelTriangleNo - 1) * 4) + \\ + subtriangleid(ranging\ from\ 0 - 3)) \end{aligned} \tag{3.1}$$

For example, if we wish to know the index in the database of the first quadrisected triangle at the first level of subdivision of the third base triangle, we would find:

normalid(28) = 20 + ((3-1)x4)+0

Two nice features of this encoding scheme are that it only requires 36 dot products for finding a representative normal amongst 27,300 in the database (20 base triangle dot products+ 4 levels*4 dot products). Since we are only interested in the normal with the smallest error at each level, we do not need to take the inverse cosine of the dot product. Encoding Lucy's 28 million triangle normals thus took 303 seconds (Table 3.5, 5th column) on a G4-500Mhz PowerPC with 1 GByte of RAM. We do not use a GPU for the results to be presented and store the database of normals used by the scene in system main memory. Timings for several models are also available in the top row of Table 3.5 in Section 3.5.5. Note that the same database produces a maximum error of not more than 2.5 degrees with all models and that the computation

time varies linearly with the size of the input model. For example, 1 million triangle normals took 9.6s (Table 3.5, 2nd column). This indicates that if we were to encode at run-time a set of normals which was half as large it would take approximately 4-5 seconds. We also note that a memory look-up of a 12-byte representative normal in system memory, or in video card memory, is always going to be faster than a bit-wise decompression that requires arithmetic. It is common in real-time systems that render models comprising hundreds of million of triangles in secondary memory only to keep a base simplified mesh level of detail (LoD) of just a few tens of thousand of triangles always present in the graphics card memory. These are then rendered asynchronously, as geometry is paged-in, and LoDs are switched [BGB+05]. According to this trend, we rendered and timed the full 69 thousand triangles that would be likely to be used in such systems rather than timing models comprised of tens of million of triangles that simply are not rendered interactively at full resolution in current systems. We did not use view-frustrum culling or LoD switching in our timings. In order to further isolate the performance of looking-up normals versus direct rendering of true normals, we also included timings of rendering without using the ATI Rage Mobility 128 graphics card for the laptop that ships with 8 Mbytes of SDRAM video memory. We compared direct (uncompressed) and indirect normals (PNORMS) results with CPU rendering. The results in Figure 3.45 show that the difference is negligible. Another nice feature is that no tolerances are required to produce a maximum error of 2.5 degrees with a five level subdivided icosahedron (Figure 3.39). Once a representative normal is chosen, the triangle gets the id of that representative normal's position in the array. In a profile mode, all the errors incurred are added and divided by the number of triangles in the model to compute the mean average error and, similarly, to calculate two standard deviations from the mean. We also kept track of the largest error incurred as the maximum error. Figure 3.39, right, shows the colour coding distribution of the errors in the normals with the maximum error in red and blue indicating small errors. In Section 3.5.5.4 we show how the colour is encoded.

Bunny	Happy	Thai	Power Plant	Lucy
69k tri. (Δ)	$1 \times 10^6\Delta$	$10 \times 10^6\Delta$	$13 \times 10^6\Delta$	$28 \times 10^6\Delta$
2.5° 0.6s	2.5° 9.6s	2.5° 95s	2.4° 110s	2.5° 303s
1.3° 3.6s	1.3° 53s	1.39° 535s	1.3° 948s	1.39° 1501s

Table 3.5: Maximum error & time for fast (top row) and accurate (bottom row.) encoding (tuned tolerance) with 2 byte normal indexing of a five times subdivided icosahedron.

3.5.4.2 Accurate encoding

In the previous section we presented a method based on offsets for hierarchically finding a representative normal in a database for a given true normal. One problem with the fast approach is the degree of symmetry between the normals of adjacent base triangles and also the symmetry between the normals of adjacent subdivided triangles derived from different base triangles. This symmetry cannot be captured with the previous approach. The problem occurs when a true normal is centered almost halfway between two or more normals in the database. The representative normals will then differ in orientation relative to the given normal by arbitrary small amounts. To cater for such symmetry, we extend the previous off-setting approach to include a tolerance test. If a normal is inside the tolerance it is added to a list of triangle normals and all the subdivided normals derived from this list will be considered too. From Figure 3.41, we observe that any horizontal line crossing the subdivided triangle can cross at most three triangles. The dihedral angle between adjacent polygons of each Platonic solid is: 138.190°, 109.471°, 70.529°, 90° and 116.565° for the icosahedron, octahedron, tetrahedron, cube and dodecahedron respectively. We use the supplementary angles in our calculations: 41.8°, 70.5°, 109.4°, 90° and 63.4° respectively. The tolerances are set as an inverted, cascaded pyramid of decreasing values. The initial dihedral angle is divided by three at each subdivision level. The final angular tolerance for a given level uses the value that is divided by three at that level and multiplies it by 1.5 also to include adjacent normals from the database at that subdivision level (Equation 3.2).

$$\begin{gathered} subdivision\ Level\ Tol = (subangle) * 1.5 \\ where\ subangle = subangle\ of\ the\ previous\ level/3 \\ for\ the\ first\ level\ of\ subdivision\ the\ subangle \\ corresponds\ to\ the\ solid's\ initial\ dehidral\ angle \end{gathered} \tag{3.2}$$

Since we are dealing with angular tolerances, we need to find the inverse cosine of the dot products. The same tolerance formula allowed us to study the numerical errors arising when the database of representative normals is derived from each of the Platonic solids. Solids that yielded distributions that tended to be more uniform within a given level of subdivision, such as the octahedron and icosahedron, were more easily accommodated by use of a single tolerance measure. The databases derived from the other solids that displayed large variations in resolution within the same level were not easily accommodated by use of a single tolerance. They required a tolerance to cover the angular symmetry of areas on the sphere where the changes in orientation were large, whilst increasing the computation in areas of small orientation steps

where a smaller tolerance would suffice. Table 3.7 shows timings obtained when we used the cascaded formula (Equation 3.2) with the Bunny model for levels 0-9. From Table 3.4 we can see that the icosahedron generates significantly more normals at each level in comparison to the octahedron and tetrahedron, a fact reflected in the timings given in Table 3.7.

We note that an arbitrary large tolerance can significantly slow encoding as more triangle normals are considered in the lists at each level. We did an exhaustive search for the angular tolerances which would optimize the speed of the accurate approach for different solids. Instead of using the cascaded tolerance formula, the timings of the accurate approach given in the bottom row of Table 3.5 used set tolerances (in radians) for levels 0-5 of: 0.05; 0.05; 0.02; 0.02; 0.01; and 0.01 respectively. Consequently the time dropped from 40.9s (Table 3.7; 5th column from left and 5th row from bottom) to 3.6s (Table 3.5; 1st column from left, and 1st row from bottom). We note that the maximum error was not affected by this optimization. For example, the Stanford Bunny model has a bounded maximum error of 1.3 degrees in Table 3.5, first column, bottom row and in Table 3.7, first column, sixth row it has a maximum error of 1.32 degrees when the cascaded tolerance formula is used. The tuned tolerances enabled the accurate method to encode 28 million triangles using five levels of subdivision of an icosahedron in 1501 seconds with the same bounded maximum error as for the Stanford Bunny of 1.3 degrees (Table 3.5; rightmost column, bottom row). In the next section we present and evaluate more results obtained by use of both encoding methods.

3.5.5 Results

In this section we evaluate some results.

3.5.5.1 Angle error analysis

Table 3.7 shows errors for encodings obtained using the databases derived by subdivision of the icosahedron, octahedron, and the tetrahedron using the accurate encoding method with the same cascaded tolerance (Equation 3.2), starting with dihedral angles of 41.8°, 70.5°, 109.4° respectively. In particular, it shows for each subdivision depth (d) the maximum error (M), the mean error (me), and the error for two standard deviations from the mean representing approximately ~98% (m2) and the encoding time in seconds. Figure 3.43 and Figure 3.44 show that the cube and dodecahedron were too irregular for subdivision and produced errors too large to be useful. Similarly, the tetrahedron produces very large and small triangles in the same mesh. This is consistent across all of its subdivision levels. We include results obtained from it in Table 3.7. Table 3.5 compares the time and maximum error of the fast encoding versus the fine tuned, accurate encoding for several models using a database derived from five subdivisions

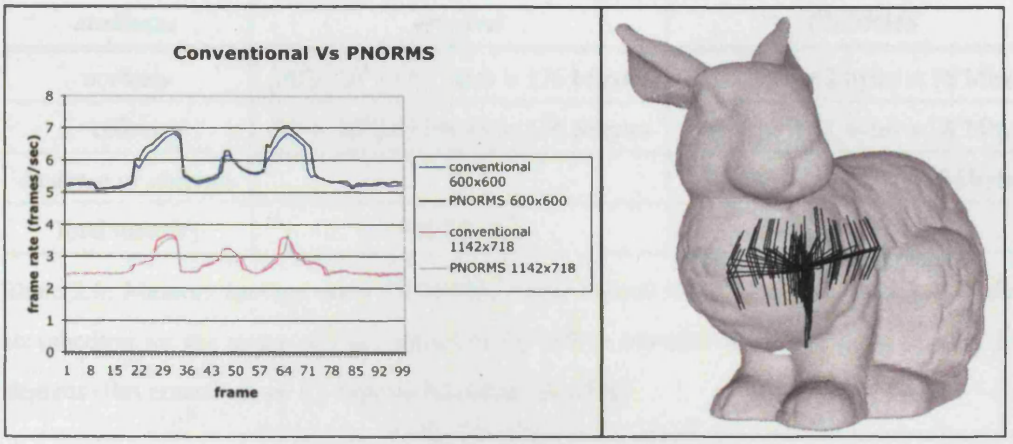


Figure 3.45: *Left:* Frame rate of direct conventional rendering of normals versus PNORMS look-up and then a rendering of the camera sequence shown in the right subimage. Both methods render 69,451 triangles in every frame. *right:* Camera view direction sequence.

of an icosahedron. It can be seen that the time varies linearly in both cases with the size of the input.

3.5.5.2 Run-time performance

We used a PowerBook G4 500 Mhz with 1 GB RAM for all the experiments in this section. Figure 3.45 shows the frame rate of direct conventional rendering of normals versus PNORMS look-up when rendering the camera sequence shown in the right image of Figure 3.45. In particular it shows that there is no significant difference in frame rate between a conventional rendering of true 12 byte normals and a run-time look-up using PNORMS whilst using the graphics card (ATI RageMobility128) for rasterization with a window size of 600x600. Figure 3.45 also shows that there was also no significant difference in frame rate between conventional rendering of true 12 byte normals and a run-time lookup using PNORMS when rendering purely in software with a window of size 1142x718.

3.5.5.3 Shading effects

Figures 3.46 and 3.47 show different shading effects produced by using representative normals of different subdivision levels of a database produced by five subdivisions of an icosahedron with the 8 million triangle normals of a scanned model of Michaelangelo's statue of David encoded using the accurate encoding method. Table 3.7 shows that the database derived from the icosahedron leads to the most accurate results. By cross-referencing the number of accessible normals for 2 byte and 4 byte indices of different solids with the errors in Table 3.7 it can be seen that the more regular icosahedron also yields the smallest representation error. If we use

attributes	*original*	*PNORMS*
normals	$(28 \times 10^6) \times 12$ bytes = 336 Mbytes	$(28 \times 10^6) \times 2$ bytes = 56 Mbytes
colour	$(28 \times 10^6) \times 12$ bytes = 336 Mbytes	$(28 \times 10^6) \times 1$ bytes = 28 Mbytes
database of normals	0	27300×12 bytes = 327 kbytes
Total memory	672 Mbytes	84.3 Mbytes

Table 3.6: Memory savings using PNORMS, 2-byte normal indexing of five times subdivided icosahedron for the statue of Lucy model of 28 million triangles for a maximum error of 2.5 degrees (fast encoding) or 1.3 degrees (accurate encoding).

accurate encoding - cascaded tolerance formula 9 subdivided PNORMS												
	icosahedron				*octahedron*				*tetrahedron*			
d	M	me	m2	t(s)	M	me	m2	t(s)	M	me	m2	t(s)
0	**37.1**	*18.1*	32	2.4	**54.6**	*31.4*	54.2	0.9	**70.1**	*37.5*	69	0.6
1	**19.2**	*9.6*	17.7	9.8	**30.3**	*16*	29.4	1.9	**45.9**	*22,9*	44.3	0.9
2	**10**	*4.7*	8.68	18.5	**16.1**	*8*	14.6	3.4	**27.4**	*10.8*	21.1	1.2
3	**5.33**	*2.31*	4.29	27	**9.27**	*4*	7.35	5.4	**29.9**	*8.88*	18.6	1.6
4	**2.69**	*1.15*	2.14	34.3	**4.87**	*2.01*	3.7	7.6	**31.1**	*7.31*	17.7	1.9
5	**1.32**	*0.58*	1.08	40.9	**3.13**	*1.03*	1.93	9.8	**32.1**	*6.82*	18.3	2.2
6	**0.66**	*0.29*	0.54	46.5	**2.49**	*0.53*	1.07	11.8	**32.7**	*6.68*	18.9	2.5
7	**0.36**	*0.14*	0.27	51.5	**2.27**	*0.28*	0.69	13.7	**33.0**	*6.6*	19.2	2.9
8	**0.18**	*0.07*	0.13	56.2	**2.19**	*0.15*	0.54	15.5	**33.2**	*6.64*	19.4	3.2
9	**0.09**	*0.03*	0.07	56.9	**2.15**	*0.09*	0.48	16	**33.3**	*6.64*	19.4	3.4

Table 3.7: Maximum error (M), mean errors(me) and the error of 2 standard deviations from the mean(m2) for the Stanford Bunny using the accurate encoding based on the icosahedron (left), octahedron (middle) and tetrahedron (right).

2 byte index normals and 1 byte for colour look-up (Section 3.5.5.4) the Lucy model requires in total 734.3 Mbytes (84.3Mb [colour+normals]+ 650 Mb [geometry]) instead of 1.3 Gbytes (672Mb [colours+normals]+650Mb [geometry]) (see Table 3.6 for details).

3.5.5.4 Colour encoding

In order to visualize the colour coding distribution of the errors in the normals of a large model such as the one shown in Figure 3.39, a 1-byte index table of 256 colours was built. A simple colour ramp function (Equation 3.3) was used to convert a single scalar value into RGB values. It was also used for adding colours to the table. In order to add 256 colours ranging from blue to red, the geterrorcolour function was called 256 times with *res* (see Equations 3.3 and 3.4) starting from 0, and incremented by 1/256 each time.

Figure 3.46: Shading effects of the Statue of David, using 2 byte normal indices of different levels of a database produced from a five times subdivided icosahedron. Normals encoded with the accurate approach. top row: from left to right: subdivision level and maximum error M in curved brackets; level 0 (M 37.3°); level 1 (M 19.3°); level 2 (M 10°); bottom row: from left to right level 3 (M 5.3°); level 4 (M 2.7°); original/true normals (M 0°).

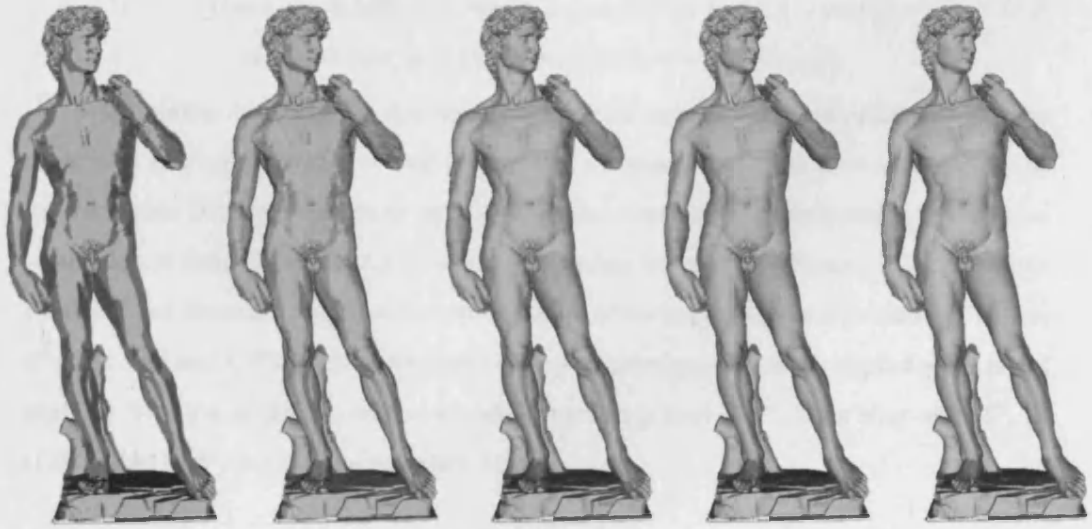


Figure 3.47: Shading effects of the Statue of David using 2 byte normal indices of different levels of a database produced from a five times subdivided icosahedron. Normals encoded with the accurate approach, from left to right: subdivision level and maximum error M in curved brackets; level 0 (M 37.3°); level 1 (M 19.3°); level 2 (M 10°); level 3 (M 5.3°); level 4 (M 2.7°); original/true normals(0°).

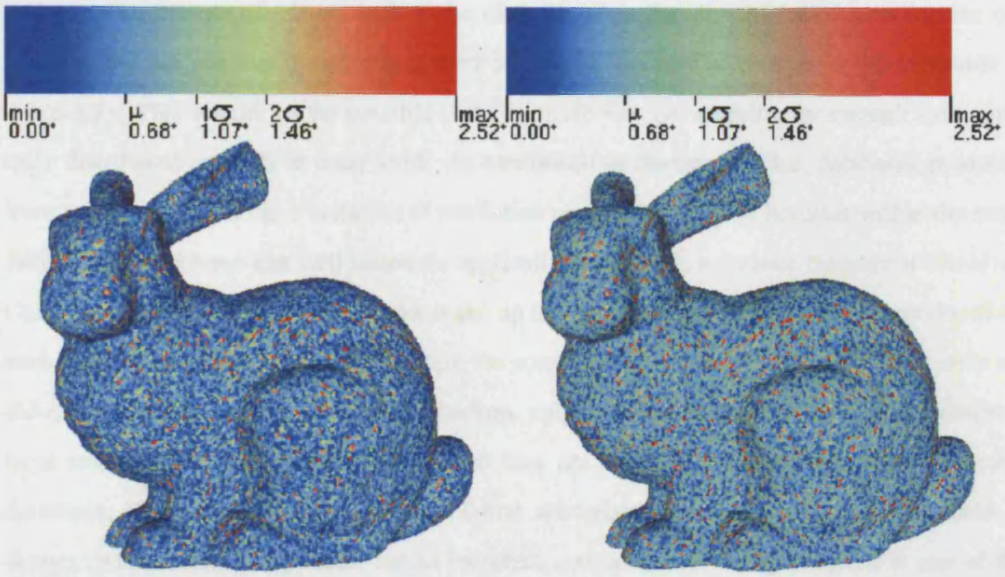


Figure 3.48: Colour ramp functions according to Equation 3.4 (left) and Equation 3.3 (right) corresponding to the errors of fast normal encoding: minimum error of 0°, mean error of 0.68°, 1σ (1.07°), 2σ (1.46°) and a maximum error of 2.52° .

$$\begin{aligned} &void\ geterrorcolour(float\ r\ float\ g\ float\ b\ float\ res) \\ &if(res <= 0.345)\{r = res;\ g = res/0.345;\ b = 1.0 - res;\} \\ &else\{r = res;\ g = (1 - res)/0.654;\ b = 1.0 - res;\} \end{aligned} \tag{3.3}$$

After normal encoding the true normal of each triangle is calculated again and the error angle with their representative normal is computed (Section 3.5.4). This error when divided by the maximum error and multiplied by 255 gives the colour index corresponding to the error. In addition to using Equation 3.3 for the colour coding we also tried Equation 3.4. However we found that Equation 3.3 yielded a colour coding of the mean that was less saturated in blue (Figures 3.48 and 3.39). Figure 3.48 shows different colour ramp functions applied to the errors resulting from use of the fast normal encoding: minimum error of 0°, mean error of 0.68°, 1σ (1.07°), 2σ (1.46°) and a maximum error 2.52°.

$$\begin{aligned} &if(res <= 0.5)\{r = res\ g = res/0.5\ b = 1.0 - res;\} \\ &else\{r = res\ g = (1 - res)/0.5\ b = 1.0 - res;\} \end{aligned} \tag{3.4}$$

3.5.6 Discussion

We would like to assess how successful our cascading tolerance formula (Equation 3.2, Section 3.5.4.2) and used in Table 3.7 was in comparing and using the databases derived from the

different Platonic solids. If we look at the mean error pattern for the icosaheron and the octahedron we can see that in both cases there is a steady halving of error (row 0 downwards in Table 3.7). This would not be possible if the formula was not catering for enough symmetrically distributed normals at each level. As mentioned in Section 3.5.4.2, databases produced from solids that had a large variation of resolution or in the density of normals within the same subdivision level were less well suited for application of a single tolerance measure at that level. Consequently a single tolerance cannot make up for the distortion of the databases produced by each solid without increasing significantly the computations in the whole level. Ultimately we did not expect good results for the tetrahedron, cube and dodecahedron owing to the inherent large and irregular triangles produced when they are subdivided. With these more irregular databases, if an error is created at one of first subdivision levels the symmetry tolerance at deeper levels cannot compensate for an incorrect normal bucket being assigned in one of the first few levels. We tried different triangle nets and patterns/tessellations for both the cube and dodecahedron but they created even more distorted triangles on subdivision than those presented for the tetrahedron. We note that the results achieved with the fast encoding algorithm presented in Figure 3.39 did not require any tolerance measure. The top row of Figure 3.46 shows images that are not dissimilar to a GIF image's 256 quantization of grey levels.

3.5.7 Conclusion

We showed that the increasing number of normals resulting from the subdivision of Platonic solids does not necessarily, in all cases, lead to a monotonic reduction in the encoding errors. We provide an alternative to bit-wise compression with the following main advantages:

- Maximum error bounds for any model. This allows normals belonging to objects to be added dynamically to a scene or to be produced from soft-deformation animation, or from large sets of user edited geometry in a VR modeling task to be encoded with confidence.

- A single hierachical-multiresolution database for a scene as a whole that allows for fast searches of representative normals during encoding without using the GPU thus freeing the graphics card to do other tasks.

- A slower but more accurate encoding method that caters for the symmetry inherent between adjacent normals of a Platonic solid.

- No reconstruction of normal ids required at run-time, again freeing the GPU to do other tasks.

We use absolute normal indices rather than bit-wise compressed indices. We showed how the number of subdivisions of different Platonic solids affects the number of normals created, the error associated with the distribution of the normals, and ultimately the length in bits that bit-wise compression methods need to use. In particular we showed that the five times subdivided icosadron has less maximum and mean error in our results, and produces 2.5 times more normals than the five times subdivided octahedron widely used in the normal compression literature. Consequently we could use absolute 16 bit normal indices with still 83% memory savings, with no normal reconstruction required.

We presented results obtained using scanned statues and the UNC power plant CAD model. We presented a fast encoding scheme that does not require tolerances and a more accurate encoding method that caters for symmetry by cascading tolerances across the levels of the database. The database generated by subdivision of an icosahedron generated the smallest maximum and mean errors of any of those obtained from the five Platonic solids. 2 byte and 4 byte databases of normals have been made available online. The 2 byte indexed database obtained from five subdivisions of an icosahedron can be used to encode accurately normals with a maximum error bound of 1.3°, with fast encoding of normals with the same database the maximum error is bound to 2.5°. The 4 byte indexed database obtained from nine subdivisions of an icosahedron can be used to accurately encode normals with a maximum error bound of <0.1°, with fast encoding of normals and using the same database of normals the maximum error is bound to 2.5° which is not lower than when using the fast encoding algorithm and a smaller database of normals obtained by a five times subdivided icosahedron. This can be explained by the fact that the fast encoding algorithm does not cater for symmetry thus the accuracy of the algorithm does not improve with larger databses of normals unlike the accurate encoding approach which does.

Our method stores normals in the database at all levels of subdivisions so as to allow hierarchical querying of new normals at run-time. This strategy is not only useful for encoding new unknown normals of modified or new geometry in the scene, but could be used to encode vectors of directional light sources or view directions for other visibility or lighting calculations (as also suggested in [BWK02]). Thus, although the current implementation of our approach does not use the graphics card capabilities, it is conceivable that algorithms could be developed using purely the 2-byte or 4-byte indices without actual decoding since there is an implicit mapping to their 3D orientations. We note that many computer games are geared to applying rigid transformations such as translation and rotation matrices essentially to unchanged representations of geometry. We hope that our method can help bring more soft deformation animation

of surfaces into applications such as games and bring memory savings when modelling/editing larger models in VR. As mentioned in Section 3.5.5.3, our compression method allowed us to load and view the 28 million triangle statue of Lucy with a laptop loaded with a maximum of 1 Gbyte of RAM, using just 739 Mbytes for the model instead of the expected 1.3 Gbytes. We found that the 256 colour table was sufficient to store all the colours of the Power Plant model.

3.6 Conclusion

We have introduced a memory-friendly octree generated in-place and without storing triangles at leaf nodes. This representation proves to be very useful for representing a scene in hierarchically manageable parts and has been incorporated in a system that automatically adjusts to the size complexity of the input mesh in order to display at a regular frame-rate the part of the model of interest, for example in an interactive editing task. With our implementation of the Octree Interaction Engine the observed frame rate variation was no more than 5-8 fps at frame rates above 20. Such a variation does not disturb the user's performance of a task such as editing [WSNR96]. The way we created the octree changes the triangle ordering with the beneficial side effect of making the mesh more coherent. To further improve mesh coherence, we plan to use the octnode ids to re-label the vertex indices as in [CRMS03]. Rendering one coarse level of detail in wireframe together with a shaded high-resolution, variable size portion of a model has proved to be an invaluable metaphor for navigating a mesh where multiple levels of detail cannot be stored. Our compression algorithm for unit normals and colour enabled us to test our Octree Interaction Engine with larger models than could previously be accommodated in main memory. The compression gave rise to no perceptible visual difference and caused no noticeable change in rendering speed.

Chapter 4

Uniform geometry reduction

4.1 Introduction

In the previous chapter an algorithm for interactive viewing and feature selection of large models was presented. These marked features can then be used to retain more detail in those areas than in the unselected parts of the model as will be shown with our non-uniform reduction framework presented in the next chapter. In this chapter we first discuss uniform LoD reduction techniques.

For scanned models, with or without texture, a simple clustered LoD version of the model was used as a navigation skin to assist model inspection. For correct feature selection an efficient depth buffer strategy was designed to ensure that the rendering of a model's original triangles was always superimposed on the rendering of the navigation skin. Garland et al. [GS97] used the quadric error reviewed in Chapter 2 to produce good quality approximations of the original model, adding and accumulating at vertex level the quadrics of edge collapse vertices. Lindstrom et al. [LT99] showed that it is possible to obtain good quality LoD meshes without having to store any quadrics. Such memory savings are particularly attractive when computing LoD representations of large models. However Lindstrom's approach optimizes volume and triangle shape making it difficult to assess any effect on quality that not storing the quadrics might have. In this chapter we make a direct comparison of the results obtained from methods in which the quadrics are stored and methods in which they are not stored. We confirm that indeed good quality approximations and in some instances better quality approximations are obtained in the approaches which were designed to operate without storing quadrics.

The choice of LoD algorithm for producing the navigation skin can be arbitrary. However, typically an analytical or numerical solution is found to establish the optimal[1] coordinates of the vertex remaining after an edge collapse operation. In the context of optimisation methods

[1] Optimal, that is, with respect to the cost function used.

based on the quadric error, problems arise in planar areas where more than one solution can exist. A determinant test or threshold on the condition of the matrix one wishes to invert is used and an alternative fall back strategy such as finding the optimal position in lower dimensions, for example, on a line, or picking the midpoint position is used instead [Gar99]. With current systems this threshold value is static and used with every model. We found that the choice of units in which a model is defined can inadvertently affect numerical results. In particular, triangles with relatively small edge lengths create even smaller value areas that when pre-multiplied by terms of a matrix make the systems of equations appear ill-posed if a suitably high threshold is not chosen. On the other hand, a threshold that is too high might make the numerical solver attempt to find optimal positions that are arbitrarily far away from the model in an ill-posed or underdetermined system.

Finding the optimal threshold for the LoD solver is a non trivial problem that can affect the overall quality of simplifications as more or less optimal positions of edge collapses are found. In the extreme, a poorly chosen tolerance can result in no matrices being inverted thereby defaulting to planar solutions such as the midpoint vertex placement. Lindstrom et al. [LT99] measure the local geometry and optimize volume and triangle shape if the normals to the planes make an angle of less than 1 degree with each other. However, patterns of features very close to planar might reflect a roughness characteristic of the material of a scanned object that one may wish to preserve in an LoD. We present an automatic solution for calibrating the condition number. We found that this automatic threshold varies from model to model with the scale of units used to represent the vertex coordinates. We compare the geometric errors resulting from a system with poorly chosen tolerance that always defaults to midpoint placement with the errors resulting when the threshold is found by our system. An implementation of the simplification method "Surface simplification using quadric error metrics" (QSlim [GS97]) and the memoryless simplification [LT99] in which quadrics are not stored was used in these experiments.

In this chapter a strategy for uniform geometry reduction is presented. The framework for non-uniform reduction presented in the next chapter is built on the concepts presented here. Numerical issues in computing the solution of an edge collapse operation are addressed in Section 4.2 and an automatic solution for condition number calibration is presented in Section 4.3. The quadric error has been rapidly adopted for its high quality results and simplification speed for both in-core and out of core models. Lindstrom's [LT99] method that does not store the quadrics in memory uses a numerical solver and the binary release of QSlim to compare his results with those of Garland. For the purposes of choosing a simplification method for building the navigation skin we compared results obtained with both methods by using the same

calibrated threshold and solver. (Section 4.7)

A framework for incorporating mesh quality constraints into any edge collapse based system that can be used in both uniform and non-uniform reduction is presented in Section 4.4. Implementation issues are briefly considered in Section 4.6. Finally a method for the treatment of border vertices is presented in Section 4.5. This method can be used in the treatment of feature vertices. Conclusions are presented in Section 4.8. Throughout this chapter, we assume that there are no duplicate (or near duplicate) vertices in the models discussed and that triangle normals are consistently oriented. We defer the problem of cleaning duplicate vertices that would otherwise generate cracks during simplification, and the problem of consistently orienting surface normals to Chapter 6.

4.2 Numerical issues

In Section 2.6 the first derivatives of the quadratic cost function(squared distance to planes) was forced to be zero and a matrix inversion used to find the optimal vertex position with the smallest summed squared distance to the planes around the vertices of the edge to be collapsed. A problem arises when the quadric of an edge collapse is constructed from several parallel planes as, in the absence of other planes, this amounts to construction of the same equation and leads to a system of fewer independent equations than unknown variables. This aspect can inadvertently create "optimal" positions arbitrarily far away from the model that appear as "spikes" on the model (right image of Figure 4.1).

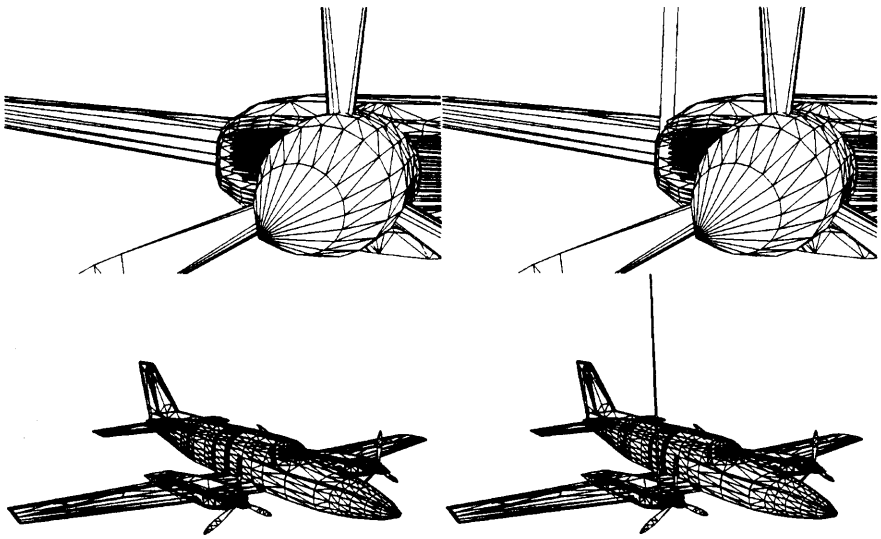


Figure 4.1: *Under determined* system; *left:* before edge collapse *right:* after edge collapse, the optimal vertex position in the flat structure lies far from the model.

The system of equations/matrix is said to be *under determined* [Str88]. Rossignac's [RB93] mesh quality constraint that detects a mesh fold-over (described below in Section 4.4.3) can somewhat alleviate this problem but unfortunately does not work in all cases, for example when the edge shared by a double sided triangle is collapsed.

One solution to this problem is to place a threshold on the determinant of the matrix defining the system of equations. If the determinant exceeds a specified tolerance other vertex placement strategies can be used, such as moving a vertex to the midpoint of an edge. Finding a suitable threshold for a particular model by trial and error can be cumbersome and is not a useful straetgy to adopt in building a software geometry reduction tool, in particular as the user may not know enough about the model in order to correct the problem.

Alternatively one can use a different solver such as *singular value decomposition* and threshold the condition number of the matrix. The condition number of a matrix can be measured by dividing the largest singular value, *wj*, by the smallest *wj* (see Press et al. [PTVF92], page 61). The authors of this book also advise thresholding to zero very small *wj* before attempting to find a solution to the equations in their implementation of singular value decomposition.

However the threshold has to be changed/the code recompiled for different models as the quadric error matrix can also become *ill conditioned* as the scale or units in which a model is expressed affects the value of the determinant. Models expressed in small units such as the Stanford Bunny with more than 34,000 vertices within a floating point sphere of radius 0.13, Figure 4.2 left) and dense scans often are comprised of very small triangles. In addition to such scale problems *per se*, the even smaller numbers associated with the areas of triangles which are used as pre-multipliers of the quadrics in order to obtain tessellation invariance (end of Section 2.6) exacerbate the problem and may even potentially make the determinant test fail at all edges even though the matrices should, in principle, be invertible. This happens for example if one tries to use a threshold that flags flat areas in a model expressed in units of a given resolution and tries to use the same *compiled* tolerance for a model with much smaller units where a higher tolerance is required. The threshold is application and model dependent. A threshold that is too high produces spikes and a threshold that is too low produces solutions such as those obtained from vertex averaging that are bound to the model's surface and yield considerably higher geometric errors than might ideally be obtained (Section 2.2).

Although a placement strategy that has been undermined so as to utilise only vertex averaging produces relatively higher errors, the fact that the quadric cost error function is still used to recompute costs and to iteratively choose the lowest cost or smallest 'damage' operation enables the resulting prioritisation of edge collapses to somewhat surprisingly still produce

models of relatively good visual quality. For example, the rendering of a simplified model produced by such a placement strategy was almost undistinguishable from that of a simplified model produced by the optimal placement strategy as shown in the middle of Figure 4.2. This suggests that the cost function has perhaps a more important role in the creation of visually good LoDs than the vertex placement strategy itself. We note that the quadric error function still penalises edge collapses in high curvature areas regardless of the vertex placement strategy. A cost function based purely on edge length and a vertex averaging placement strategy is blind to curvature and, as the above would indicate, results in visually poorer LoDs as can be seen around the area of the ears on the Standford bunny model in Figure 4.2 right.

We present in the following section an algorithm that automatically finds a suitable condition number for the LoD solver for each model, avoids spike artefacts and ensures better geometric errors than schemes which default inappropriately to a vertex midpoint placement strategy.

Another solution to this problem is to measure the angle of the geometry associated with the edge collapse. Lindstrom [LT99] uses different placement strategies if the angle between the triangle planes connecting to the vertices of the edge to be collapsed is below 1 degree. A highly tessellated sphere is locally flat everywhere and areas of scanned models such as the underside of the Stanford bunny (Figure 4.3 top left) may appear to be flat. However, as will be seen in the next subsection, in such cases the equations can be solved for the optimal vertex positions as the matrices are in fact perfectly invertible. Depending on the application intended for the simplifications, whether it be numerical simulation or the visualization of museum artifacts for example, a completely flat simplification is often less desirable than approximating the roughness present in the original model.

4.3 Automatic condition number calibration

In this section an automatic solution to the problem of choosing a suitable tolerance for the condition number of the matrices for simplification is presented. The solution in which we build a graph of condition numbers is presented in Section 4.3.1. The relation between curvature and the condition numbers is studied in Section 4.3.2. The effect of pre-multiplying the quadric elements by the triangle areas for tessellation invariance is shown in Section 4.3.3. The shape of the condition number curve is studied for models of different characteristics in Section 4.3.4 and finally the same calibrated condition number is used to compare the geometric error obtained by means of QSlim and Memoryless simplification in Section 4.7.

4.3.1 Building the log graph

A solution to the problem of choosing an automatic condition number tolerance is to use SVD to calculate the condition numbers associated with all the possible edge collapses and to sort the condition numbers in order of increasing value. A 2D graph can then automatically be created in which the condition number values are plotted as the ordinate and the abscissa is the rank order of the sorted condition numbers. To better visualize the variation of the condition number values, the graph is built on a log scale for the ordinate. As will be seen for the variety of models studied in this section, typically the resulting curve's few last rightmost values increase sharply indicating the presence of a few less well conditioned matrices. This turning point is normally high enough a threshold to allow for the great majority of the matrices in the model that are well conditioned to be inverted.

To build the graph, one can find the maximum condition number, and scale the values according to the maximum amount of vertical pixels available, likewise with the horizontal axis and the number of condition entries. Prior to any mesh simplification a *scree-test* [Cat66] is used automatically to determine the condition number threshold. To carry out the scree test the chord from the first, lowest of the ranked condition numbers to the last, highest value is constructed. The point on the condition number curve lying at the greatest perpendicular distance below and to the right of this chord is chosen as the threshold (*wthres*) (red line in Figure 4.3, right)).

4.3.2 Curvature and condition number correlation

To study the relation between curvature and condition numbers in the quadric error approach the curvature was measured using the same approach used in Fei [FCGW02] as introduced in Garland's thesis [Gar99].

Figure 4.3 shows that there is a correlation between well condition matrices (blue areas in middle column) and numerical variation in curvature (areas with even only slight changes in colour in the left column). Ill-conditioned matrices (red areas of middle column) are associated with areas of low curvature (large homogeneous dark red area at the base of the scanned statue in the bottom of the left column). In particular, there is enough numerical variation in the vertex positions on the base of the Stanford bunny model (top left figure shows several variations in colour/curvature) to indicate that the matrices at the base are invertible (blue in top middle figure), with the exception of a few nearly planar areas near the hole boundaries. The population of high value condition numbers in the top right is thus very small.

Once the condition number threshold has been determined (see red line in Figure 4.3, right), all matrices associated with edge collapses with a condition number below the value of the red line can be inverted. Matrices with condition numbers above the value of the red

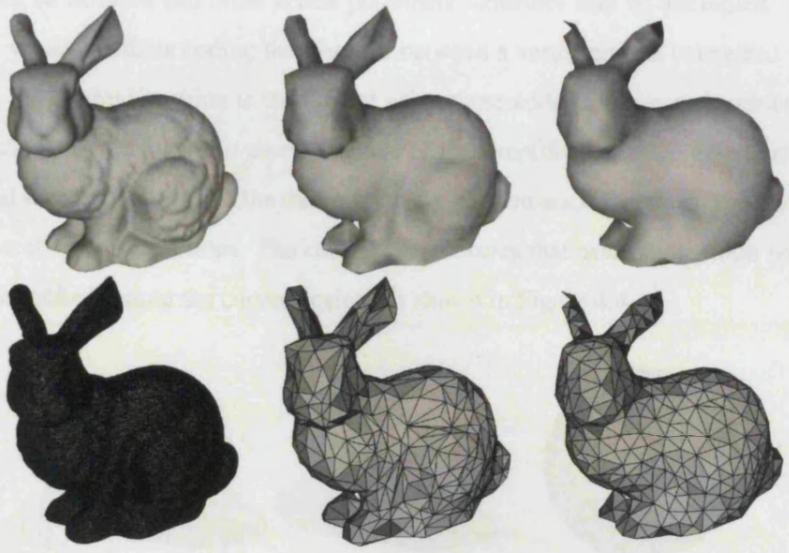


Figure 4.2: *Fine tuned condition number tolerance; left:* original Stanford bunny model, 69,451 triangles*; centre:* Lindstrom's quadric based cost, fine tuned tolerance, 1,000 triangles; *right:* edge length cost, vertex average placement/surface bound solutions, 1,000 triangles.

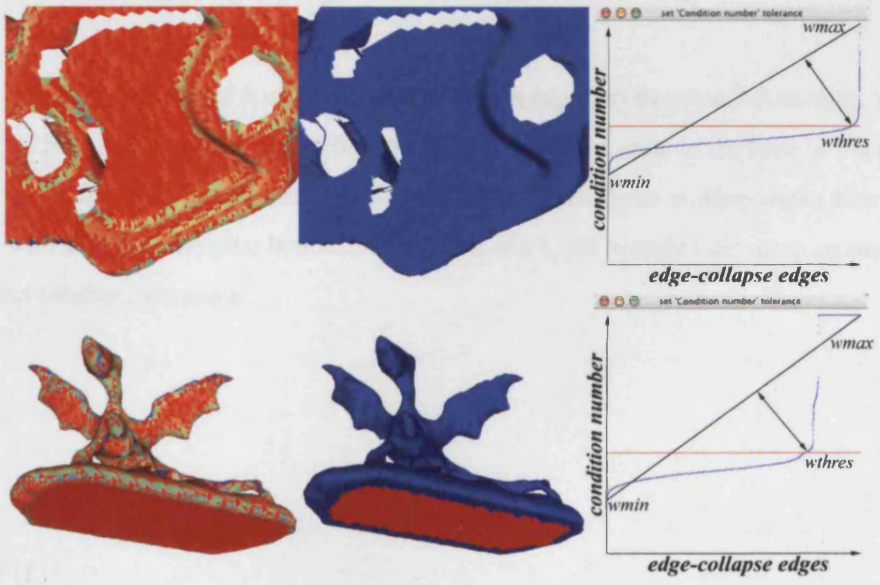


Figure 4.3: *Curvature and condition number correlation; left:* curvature colour coding (planar areas are dark red, high curvature areas are light green); *middle:* condition number colour coding (red areas have a value close to the maximum condition number found, areas in blue have 'healthy' condition numbers); *right:* top: *wthres*=2.2x10^9; *wmax*=2.3x10^14;*wmin*=795; bottom: *wthres*=5.2x10^6; *wmax*=6x10^15; *wmin*=3642.

line will not be inverted and other vertex placement strategies will be attempted. In this case temporary vertex quadrics coding the distance between a vertex and its connected vertices are computed, the optimal position is then found using these added temporary quadrics. The temporary quadrics are then used to assess the cost of the simplification, the reciprocal of the cost is taken and the result negated. One thus effectively has two error functions that are being optimised under different conditions. The construction ensures that near-planar areas with negative costs are simplified before the curved regions as shown in Figure 4.4 a).

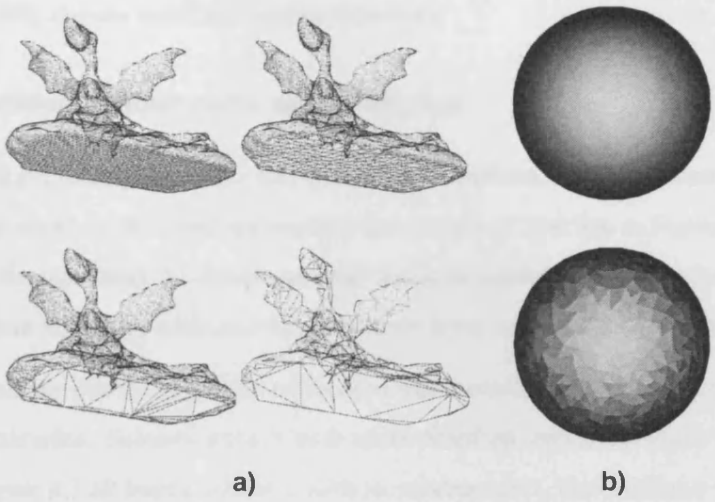


Figure 4.4: *Simplification of both ill and well condition edges; a)* from top, left to right: 11,258, 9,256, 6,256, 1,156 triangles; simplification of ill conditioned areas at the base of the model, from top, *b)* top: well conditioned sphere with 1,310,720 triangles making angles less than 1 degree with adjacent triangles; bottom: flat shading of a 1,000 triangle LoD using an automatic condition number calibration.

4.3.3 The effect of the scale of triangle areas on the conditions numbers

As described in [Gar99] one can achieve tessellation invariance during simplification if one pre-multiplies the fundamental quadric constructed from a particular triangle by a third of the triangle's area before adding the quadric to the sum to be accumulated as the vertex quadric. Figure 4.5 shows that the overall shape of the condition number curve remains more or less unchanged when the vertex quadrics are constructed with this area weighting but that the condition numbers are themselves all increased considerably. The red lines in each graph shows the automatically chosen condition number thresholds.

4.3.4 Condition number curve shape variation

As mentioned previously, the scree test chooses the condition number furthest to the bottom right from the chord of the condition number curve (*wthres*) (red line in Figure 4.6, b)). Were the absolute distance from the chord used, the threshold chosen would have been much lower than appropriate leading to inversion of significantly fewer matrices. (red line in Figure 4.6, a)).

The automatic condition number calibration was succesfully tested on several objects with varied characteristics. Scanned models such as the Stanford bunny, the happy buddha and the dragon of Figure 4.3 all have a similar condition number curve, that increases slowly and then tails off to high values towards the right of the curve.

On comparing the curves of the Stanford bunny and the dragon model, one can observe that there is a difference between the condition numbers values on what appears to be artifically created flat areas on the base of the dragon, from the values on the approximately flat scanned area in the base of the bunny. Although the number of triangles on the two models are different, the proportion of high condition numbers is disproportionately much smaller in the case of the bunny model. Variation in the vertex positions in the base of the bunny could be due to scanner noise or could reflect the roughness/texture of the material of the original model.

Computer generated models, whether they are procedural or manually created produce condition number curves with a smaller range (*i.e.* less variation in the condition numbers). A subdivided icosahedron/sphere, a cube and the Epcot model are shown in Figure 4.7. The quadrics of the cube are well defined and constructed from the fundamental quadrics of planes making angles of 90 degrees with each other. There are thus only two condition number for the cube, reflecting the polygon nets illustrated in Section 3.5.3.2, Figure 3.42. The condition numbers for the significantly tessellated sphere are high, reflecting local flatness and the relatively small areas of the triangles used in the area weighting of the quadric error terms.

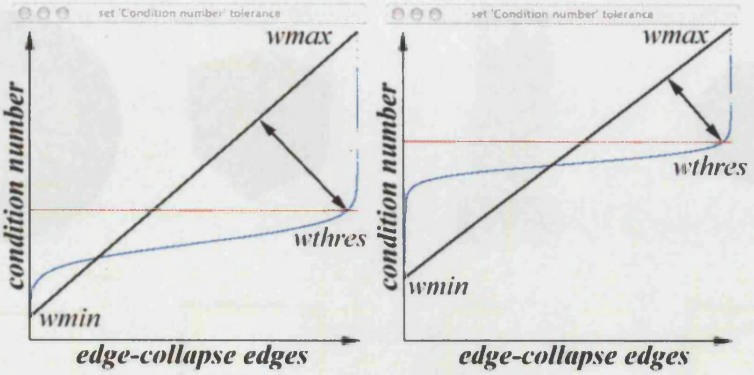


Figure 4.5: *The effect of areas/tessellation invariance on condition numbers of the Stanford bunny model; left:* quadrics are not constructed by pre-multiplication by triangle areas; *wthres*=5679.6 *wmax*=8.6x10^8 *wmin*=4.8; *right:* quadrics are constructed by weighting by triangle areas; *wthres*=2.2x10^9; *wmax*=2.3x10^14; *wmin*=795.

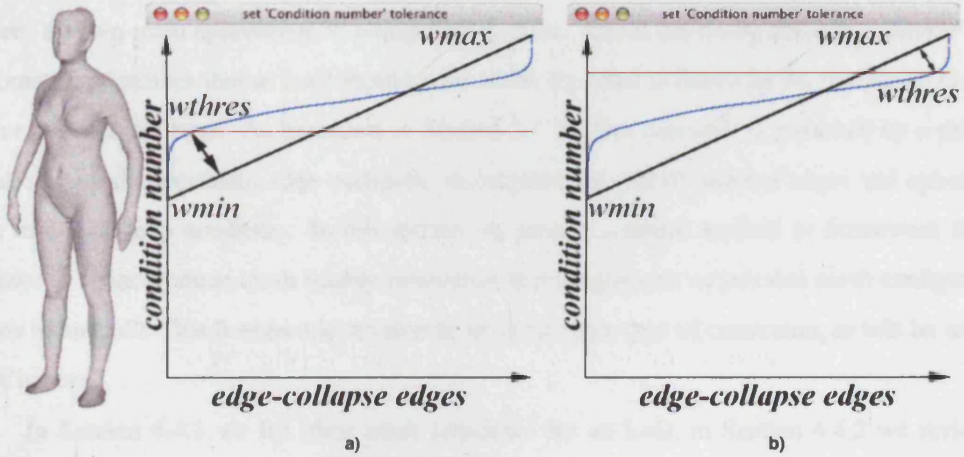


Figure 4.6: *Automatic condition number calibration of the Ciara model (135,192 triangles) wmax*=1.4x10^13 *wmin*=7.9x10^6 - *a) condition threshold wthres*=3.2x10^9 *detected using the highest absolute distance b) condition threshold wthres*=5.8x10^11 *detected using the highest signed positive distance.*

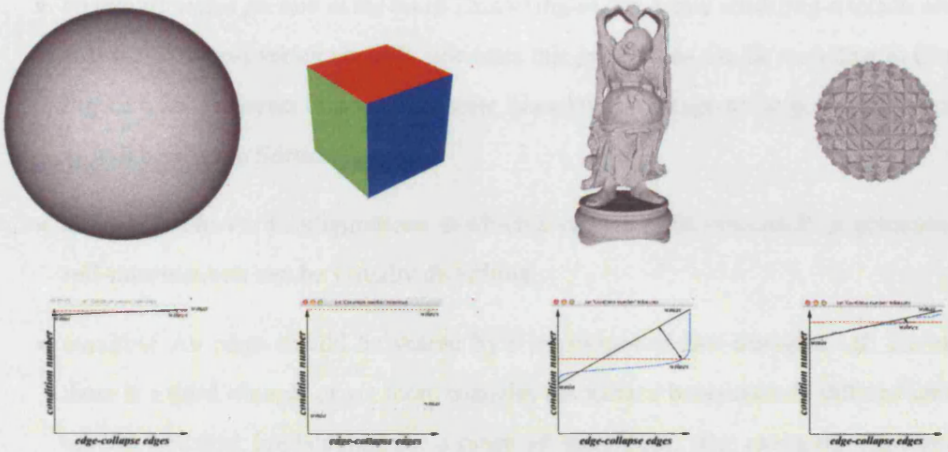


Figure 4.7: *Condition number characteristics; top:* 1.3 million triangle sphere of radius 1.0 (*wthres*=9.5x10^9; *wmax*=1.3x10^10; *wmin*=4.2x10^9), 12 triangle cube(*wthres*=6.5; *wmax*=6.5; *wmin*=5.5), 1 million triangle happy buddha statue(*wthres*=1.8x10^11; *wmax*=9.6x10^18; *wmin*=2.6x10^7), 1,536 triangle model Epcot(*wthres*=114; *wmin*=53; *wmax*=186) ; *bottom:* condition number graphs

4.4 A framework for incorporating mesh quality constraints in an LoD system

There are two main approaches to geometry reduction. One is driven by the target number of geometric primitives that an LoD should have whilst the other is driven by the maximum pixel error an LoD can have. As described in Section 2.1 the first approach is governed by a decimation loop that performs edge collapses, recomputes the cost of affected edges and upholds the min-max heap condition. In this section we present a simple method or framework that allows us to incorporate mesh quality constraints that can prevent undesirable mesh configurations in an LoD. This framework can also be used for other type of constraints as will be seen in Chapter 5.

In Section 4.4.1 we list ideal mesh properties for an LoD, in Section 4.4.2 we review Gueziec's triangle aspect ratio constraint [Gue96] and in Section 4.4.3 we review Rossignac's mesh fold-over test [RB93]. Finally, in Section 4.4.4 we show how these constraints can be easily incorporated in the decimation loop of an LoD system.

4.4.1 Ideal mesh properties

Ideally an LoD would have the following properties:

- *no thin triangles present in the mesh* Thin triangles can create rendering artefacts although area weighting of vertex normals alleviates this problem as can be seen later in Chapter 6 Figure 6.25. However thin or degenerate triangles can compromise geometric algorithms as will be seen in Section 6.1.3.5.

- *no mesh foldovers* Configurations in which a surface folds onto itself or generates mesh self-intersections can be visually disturbing.

- *manifold* An edge should be shared by a maximum of two triangles. If, for example there is a third triangle or yet more triangles the surface becomes non-differentiable with several practical implications for a range of algorithms. For example, algorithms that attempt consistently to orient surface normals according to a surface based approach will then fail as shown in Section 6.1.2.3.

- *semi-regular* The edge valence, *i.e.* the number of edges connecting to a vertex should be approximately constant in a well-formed mesh.

- *C1, C2 smoothness* As discussed in Section 2.4.1 adjacent triangles that share the same vertex indices, e.g. that have no gaps between them are C0 continuous. In addition to C0 if a surface's first and second derivative is continuous the surface is said to be C1 and C2 smooth respectively. Discrete meshes generally are neither C1 nor C2. However this can be achieved by using and applying an appropriate subdivision surface representation to the mesh [Sta00, Kob97].

In the following sections we describe how the first three constraints can easily be implemented. The fourth constraint has an important role managing the interface areas of high and low detail areas in the non-uniform LoD system presented in the next chapter.

4.4.2 Triangle aspect ratio constraint

Andre Gueziec's triangle fairness function [Gue96] provides a measure for a triangle's aspect ratio:

$$c = \frac{4\sqrt{3a}}{l_0^2 + l_1^2 + l_2^2} \tag{4.1}$$

where domain $c \in [0, 1]$, 0 being the value for a thin triangle, and 1 the value for an equilateral triangle. In the above, a is the triangle area, and l_0^2, l_1^2, l_2^2 the squared length of the three edges of the triangle.

As mentioned above thin triangles can lead to rendering artefacts, and create problems for geometric algorithms. We study the spatial distribution of thin triangles in the context of a ray

firing strategy for consistently orientating the triangles of an object. In Chapter 6 (Figure 6.24) we used Gueziec's formula together with a modification to Equation 3.3 that maps the aspect ratio value to a colour. In Chapter 3 the value 1.0 is mapped to red as the maximum error, but in the context of triangle aspect ratio visualization the value 0 is mapped to red, indicating a thin triangle. As can be seen in Figure 4.8 the chromaticity diagram follows a tilted quadratic curve. To compensate for this tilt the cut off point in the ramp function of Equation 3.3 is 0.345 instead of 0.5.

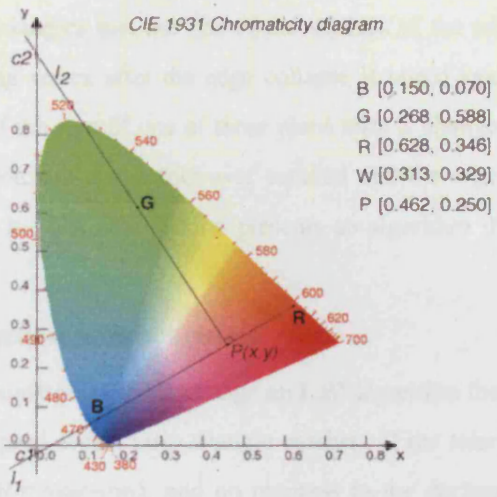


Figure 4.8: CIE 1931 Chromaticity diagram and colour mapping

if(res<=0.345) {r=1.0-res, g=res/0.345, b=res}

else {r=1.0-res g=(1-res)/0.654; b=res}

If an edge collapse produces triangles with an aspect ratio close to zero the operation is penalized. However, if the triangles were already thin before the attempted edge collapse, we allow the collapse to occur in a bid that the mesh will improve the aspect ratio as the triangles increase in size. Our function *does_mesh_thin(c)* measures the aspect ratio of triangles before the edge collapse and afterwards. If one of the triangles has an aspect ratio of 0.16 or less before the collapse we allow the collapse regardless of the aspect ratio of triangles after the collapse. The value of 0.16 was chosen by empirically measuring the triangle aspect ratio of typically thin triangles in a mesh (see Figure 5.2 in the next chapter). If initially the triangles were all

greater than 0.16, then the aspect ratio of the triangles after the collapse is compared with the set tolerance and, if the aspect ratio is smaller than the tolerance, a penalty will be enforced to discourage the edge collapse.

In the next section we review the mesh-fold over test.

4.4.3 Mesh fold-over constraint

Jarek Rossignac presented a simple and effective way to detect a potential mesh fold-over caused by an edge collapse operation. Specifically, planes perpendicular to the surface are placed at edges that do not connect directly to either of the vertices of the edge being collapsed. These edges belong to triangles that use one of the vertices of the edge to be collapsed. The position of the remaining vertex after the edge collapse is tested against the network of perpendicular planes and, if the sign of one of those plane tests is changed relative to what it was before the collapse, a potential mesh fold-over occured and the edge collapse should have a penalty associated with it. The next section presents an algorithm that can incorporate such mesh quality tests.

4.4.4 Uniform reduction LoD heuristics

A triangle aspect ratio constraint cannot change an LoD algorithm that generates poor meshes into one that produces good aspect ratio triangle meshes. If the tolerance is set too high, the system can become over-constrained, and no progress in the decimation is made. To avoid this situation, a deadlock mechanism keeps track of the number of attempts carried out. If the number of attempts reaches the point where it exceeds the number of edges in the mesh, the edge is collapsed. Algorithm 3 shows the mesh quality constraints integrated in the decimation loop of a uniform reduction LoD system.

In the next section we derive the "border quadric", which provides a way of simplifying edge borders. The technique is also used in the next chapter in our non-uniform reduction of features framework. One of the 3D models used to study the performance of different simplification algorithms under our non-uniform strategy is a face scan comprised of a significant number of border edges. The border quadric ensures the equal treatment of border edges when comparing different aspects of the various algorithms.

4.5 Border Quadric

Often 3D models have holes or borders that need to be addressed in the context of geometry reduction. If the holes are not addressed, the borders can recede or the holes close and the topology of the model may be changed [RR96]. Garland [Gar99] detects borders and adds the fundamental quadric of a perpendicular plane to the quadrics of border vertices. The quadric at

Algorithm 3 Uniform reduction LOD heuristics and mesh quality constraints.

```
function decimate ()
    deadlock = 0
    deadlocklimit = #edges
    #target_triangles = #triangles-decimation_rate
    while(#triangles > #target_triangles)
        c=h->get_heap[0]// get top of heap (smallest cost edge collapse)
        go=1
        if(#edges == 0) // no edges left
            print "done!" break
        else
            if(does_mesh_foldover(c)) // does the mesh foldover?
                go=0
            if(go)
                if(does_mesh_thin(c))// thin triangle generated?
                  go=0
            if(go)
                deadlock=0
                do_edge_collapse(c)
            else
                if(deadlock==deadlocklimit)// deadlock-
                  deadlock=0 // -limit reached, allow progress
                  do_edge_collapse(c)
                else
                  deadlock=deadlock+1
                  c2=h->get_heap[#current_edges-1]
                  h->re_insert(c,c2.cost+1000)
```

these border vertices is further multiplied by a value of 1000 and by the length of the edge for tessellation invariance. Lindstrom constrains the solution to lie on a perpendicular plane at border edges, without over-constraining the simplification [LT99]. Hoppe presented the concept of preserving the geometry of discontinuity curves [Hop96] in which boundaries between different colour attributes, boundaries along creases, or corners in scalar attributes and borders were treated in the same manner by constraining and projecting points to the discontinuity curves. Similarly we found that the treatment of border edges can also be utilised when managing different buffer regions in the context of our non-uniform geometry reduction framework as will be seen in the next chapter.

In this section we present the algebra required to constrain a solution to lie in a perpendicular plane. We note that constraining a solution to the perpendicular plane has steps in common with constraining a solution to lie along a line. We first present the formulation to find a solution along a general line in Section 4.5.1 then the formulation is expanded to form the border quadric placement and is presented in Section 4.5.2.

The border quadric is used later in this chapter when simplifying the open surface face scan when comparing different LoD methods with the same calibrated solver in Section 4.7.

4.5.1 Optimal along a line:

As mentioned in Section 4.3 in the context of the quadric error, reviewed in Chapter 2.6, there are geometric configurations in a 3D model for which a single optimal solution for the vertex location does not exist. For example, if all the triangle normals around x_1 and x_2 are parallel then the solution for the new vertex position could lie anywhere on a plane. In this situation, one could attempt to constrain the solution to lie on a line between x_1 and x_2 as in [Gar99, LT99]. Figure 4.9 shows the explicit parameterisation of the solution x_s when it is constrained to lie on the line segment between the two endpoints of the edge (x_1,x_2) to be collapsed.

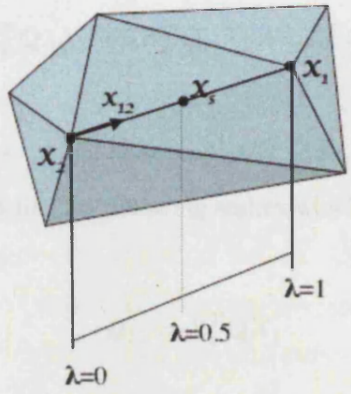


Figure 4.9: The solution we wish to find x_s is constrained to the to lie on the line between the end points of the edge (x_1 , x_2) to be collapsed. Setting $\lambda = 0.5$ would be equivelant to the linear placement method of the average position of the endpoints.

We recall that in homogeneous notation:

$$Qerror = X^T \mathbf{Q} X \tag{4.2}$$

where:

$$X = \begin{pmatrix} x \\ 1 \end{pmatrix} ; \quad X_1 = \begin{pmatrix} x_1 \\ 1 \end{pmatrix} ; \quad X_2 = \begin{pmatrix} x_2 \\ 1 \end{pmatrix} . \tag{4.3}$$

We substitute X for an interpolated solution along the line between $x_1$and x_2

$$x = \lambda x_1 + (1 - \lambda) x_2$$

where $0 \leq \lambda \leq 1$or equivalently:

$$X = \lambda X_1 + (1 - \lambda) X_2. \tag{4.4}$$

To keep the algebra symmetric, one can rewrite Equation 4.4 with the following notation:

$$X = \lambda X_1 + \mu X_2 \tag{4.5}$$

where $\lambda + \mu = 1$, and $0 \leq \lambda, \mu \leq 1$. On substituting Equation 4.5 into Equation 4.2 we obtain:

$$Qerror = [\lambda X_1 + \mu X_2]^T \quad \mathbf{Q} \quad [\lambda X_1 + \mu X_2] \tag{4.6}$$

i.e.:

$$Qerror = \lambda X_1^T \mathbf{Q} \lambda X_1 + \lambda X_1^T \mathbf{Q} \mu X_2 + \mu X_2^T \mathbf{Q} \lambda X_1 + \mu X_2^T \mathbf{Q} \mu X_2 \tag{4.7}$$

$$Qerror = \lambda^2 X_1^T \mathbf{Q} X_1 + 2\lambda\mu X_1^T \mathbf{Q} X_2 + \mu^2 X_2^T \mathbf{Q} X_2 \tag{4.8}$$

It is now convenient to define the following scalars which are easily computable from the vertex coordinates X_1 and X_2:

$$Q_{11} = X_1^T \mathbf{Q} X_1$$

$$Q_{12} = X_1^T \mathbf{Q} X_2 \equiv X_2^T \mathbf{Q} X_1 = Q_{21}$$

$$Q_{22} = X_2^T \mathbf{Q} X_2 \tag{4.9}$$

In terms of these scalars, Equation 4.8 becomes:

$$Qerror = \lambda^2 Q_{11} + 2\lambda\mu Q_{12} + \mu^2 Q_{22} \tag{4.10}$$

It is now easy to substitute $\mu = 1 - \lambda$:

$$Qerror = \lambda^2 Q_{11} + 2\lambda\ (1-\lambda)\ Q_{12} + \ (1-\lambda)^{\ 2} Q_{22}$$

i.e.:

$$Qerror = \lambda^2 Q_{11} + 2\lambda Q_{12} - 2\ \lambda^2\ Q_{12} + Q_{22} - 2\lambda Q_{22} + \ \lambda^2\ Q_{22}$$

which implies:

$$Qerror = \lambda^2\ [Q_{11} - 2Q_{12} + Q_{22}]\ + 2\lambda\ [Q_{12} - Q_{22}]\ \ + Q_{22} \tag{4.11}$$

Since Q is the sum of positive semi-definite terms it should have a unique minimum, given by:

$$\frac{\partial Qerror}{\partial \lambda} = 0$$

from which it follows that:

$$2\lambda\,[Q_{11} - 2Q_{12} + Q_{22}] + 2\,[Q_{12} - Q_{22}] = 0$$

and thus that:

$$\lambda = \frac{Q_{22} - Q_{12}}{Q_{11} - 2Q_{12} + Q_{22}} \tag{4.12}$$

4.5.2 Border quadric: optimal along a plane

This is similar to the method used when x_s was constrained to lie on the line joining x_1 and x_2. Here x_s lies on a plane so we need a pair of stable basis vectors in the plane in terms of which one may parameterize x_s. Given that we have two points x_1 and x_2 lying in the plane and its normal $A = (\alpha, d)^T$(say), the obvious choice is the local orthonormal basis shown in Figure 4.10. The Figure 4.10 shows the explicit parameterisation of the solution x_s. Given the two endpoints of the edge (x_1 , x_2) to be collapsed we form a local orthonormal basis using the fact that x_1 and x_2 lie on the constraint plane α which is perpendicular to the average normal of the triangles meeting at x_1 and x_2.

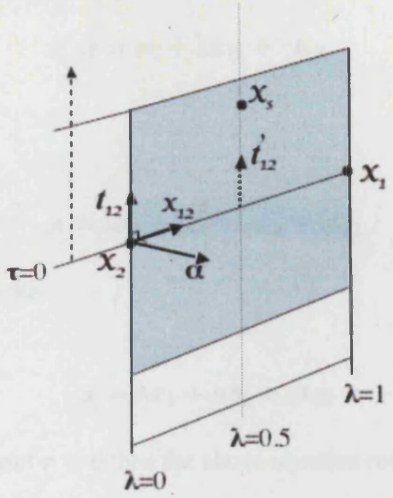


Figure 4.10: Orthonormal basis of the border quadric on the edge (x_1,x_2) to be collapsed. The solution we wish to find x_s is constrained to lie on a plane perpendicular to the average normal of the triangles meeting at x_1 and x_2. The solution is shown to be further constrained to lie along the line $\lambda = 0.5$.

The constraint plane A is represented by:

$$ax + by + cz + d = 0; \quad \alpha = (a, b, c); \quad A = (\alpha, d)$$

Both the endpoints (x_1 , x_2) and the solution x_s lie on the plane so:

$$A^T X_1 \equiv A^T X_2 = 0 \tag{4.13}$$

$$A^T X_s = 0 \tag{4.14}$$

From Figure 4.10 one can see that x_{12} is a displacement vector with zero fourth component in homogeneous co-ordinate form:

$$x_{12} = x_1 - x_2$$

the t_{12} is also a displacement vector resulting from the cross-product according to the right hand rule from the constraint plane α and the displacement vector x_{12} :

$$t_{12} = \alpha \times x_{12}$$

Thus, for any x in the plane,

$$x = x_2 + \lambda x_{12} + \tau t_{12}$$

or equivalently:

$$x = \lambda x_1 + (1 - \lambda) x_2 + \tau t_{12} \tag{4.15}$$

If $\lambda + \mu = 1$, we can write:

$$x = \lambda x_1 + \mu x_2 + \tau t_{12} \tag{4.16}$$

We note that if we demand $\tau = 0$ then the above equation reduces to the previous approach of finding the optimal vertex location along a line. As previously we utilize in homogeneous matrix notation:

$$X_{12} = \begin{pmatrix} x_{12} \\ 0 \end{pmatrix} \tag{4.17}$$

and by analogy define:

$$T_{12} = \begin{pmatrix} t_{12} \\ 0 \end{pmatrix} \tag{4.18}$$

The homogeneous vector form of Equation 4.16 is then with $\lambda + \mu = 1$ as above.

$$X = \lambda X_1 + \mu X_2 + \tau T_{12} \tag{4.19}$$

If we now recall that:

$$Qerror = X^T \mathbf{Q} X$$

we can substitute for X from Equation 4.19 and minimize the quadric Q. Thus, we find:

$$Qerror = [\lambda X_1 + \mu X_2 + \tau T_{12}]^T \quad \mathbf{Q} \quad [\lambda X_1 + \mu X_2 + \tau T_{12}]$$

or:

$$Qerror = \lambda X_1^T \mathbf{Q} \lambda X_1 + \lambda X_1^T \mathbf{Q} \mu X_2 + \lambda X_1^T \mathbf{Q} \tau T_{12} + \mu X_2^T \mathbf{Q} \lambda X_1 + \mu X_2^T \mathbf{Q} \mu X_2 + \\ + \mu X_2^T \mathbf{Q} \tau T_{12} + \tau T_{12}^T \mathbf{Q} \lambda X_1 + \tau T_{12}^T \mathbf{Q} \mu X_2 + \tau T_{12}^T \mathbf{Q} \tau T_{12} \tag{4.20}$$

For convenience, as previously we use the labels:

$$Q_{11} = X_1^T \mathbf{Q} X_1$$

$$Q_{12} = X_1^T \mathbf{Q} X_2 \equiv X_2^T \mathbf{Q} X_1 = Q_{21}$$

$$Q_{22} = X_2^T \mathbf{Q} X_2$$

and introduce the following additional labels:

$$Q_{T1} = X_1^T \mathbf{Q} T_{12}; \quad Q_{T2} = X_2^T \mathbf{Q} T_{12}; \quad Q_{TT} = T_{12}^T \mathbf{Q} T_{12} \tag{4.21}$$

On substituting labels 4.9 and 4.21 into Equation 4.20 we have:

$$Qerror = \lambda^2 Q_{11} + 2\lambda\mu Q_{12} + \mu^2 Q_{22} + 2\lambda\tau Q_{T1} + 2\mu \ \tau Q_{T2} + \tau^2 Q_{TT} \tag{4.22}$$

$$= \lambda^2 Q_{11} + 2\lambda(1-\lambda) Q_{12} + (1-\lambda)^2 Q_{22} + 2\lambda\tau Q_{T1} + 2(1-\lambda)\tau Q_{T2} + \tau^2 Q_{TT}$$

$$= \lambda^2 Q_{11} - 2\lambda^2 Q_{12} + 2\lambda Q_{12} + \lambda^2 Q_{22} + Q_{22} - 2\lambda Q_{22} + 2\lambda\tau Q_{T1} + 2\tau Q_{T2} - 2\lambda\tau Q_{T2} + \tau^2 Q_{TT}$$

$$= \lambda^2 [Q_{11} - 2Q_{12} + Q_{22}] \quad + 2\lambda [Q_{12} - Q_{22}] + Q_{22} + 2\lambda\tau \quad [Q_{T1} - Q_{T2}] + 2\tau Q_{T2} + \tau^2 Q_{TT} \tag{4.23}$$

Since $Qerror$ or simply Q is a positive semi-definite quadratic form 4.23 should have a unique minimum that may be obtained by setting $\frac{\partial Q}{\partial \lambda} = 0$ and $\frac{\partial Q}{\partial \tau} = 0$. Thus:

$$\frac{\partial Q}{\partial \lambda} = 2\lambda\left[Q_{11} - 2Q_{12} + Q_{22}\right] + 2\left[Q_{12} - Q_{22}\right] + 2\tau\left[Q_{T1} - Q_{T2}\right] = 0 \tag{4.24}$$

$$\frac{\partial Q}{\partial \tau} = 2\tau Q_{TT} + 2Q_{T2} + 2\lambda\left[Q_{T1} - Q_{T2}\right] = 0 \tag{4.25}$$

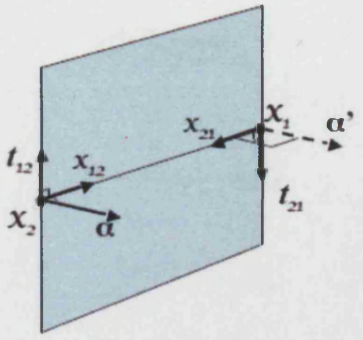


Figure 4.11: Border quadric

From Figure 4.11 one can see that:

$$x_{21} = x_2 - x_1$$

$$X_{21} = \begin{pmatrix} x_{21} \\ 0 \end{pmatrix} \tag{4.26}$$

and:

$$t_{21} = \alpha \times x_{21}$$

We let:

$$T_{21} = \begin{pmatrix} t_{21} \\ 0 \end{pmatrix} \tag{4.27}$$

and note that it is also true that:

$$X_{12} = -X_{21}$$

$$T_{12} = -T_{21}$$

Thus, the terms $Q_{T1} - Q_{T2}$ appearing in Equation 4.24 and Equation 4.25 may be written as:

$$\begin{aligned} Q_{T1} - Q_{T2} &= X_1^T Q T_{12} - X_2^T Q T_{12} \\ &= T_{12}^T Q (X_1 - X_2) \\ &= T_{12}^T Q X_{12} \\ &= -T_{21}^T Q (-X_{21}) \end{aligned}$$

Thus:

$$Q_{T1} - Q_{T2} = T_{21}^T Q X_{21} \tag{4.28}$$

The right hand side of Equation 4.28 is thus independent of the order of the co-ordinates and we may therefore in a shorthand notation write:

$$Q_{T1} - Q_{T2} = Q_{TX} = Q_{XT} \tag{4.29}$$

with the last relationship following from the fact that Q is a symmetric matrix. For completeness, we also note, from the algebra above that:

$$Q_{TX} = T_{12}^T Q X_{12} = T_{21}^T Q X_{21} \tag{4.30}$$

If we use the above shorthand notation, Equation 4.25 becomes:

$$\tau Q_{TT} + Q_{T2} + \lambda Q_{TX} = 0$$

Thus, provided the scalar $Q_{TT} \neq 0$:

$$\tau = -\frac{1}{Q_{TT}} \left[Q_{T2} + \lambda Q_{TX} \right] \tag{4.31}$$

If we now substitute into Equation 4.24 we find:

$$\lambda \left[Q_{11} - 2Q_{12} + Q_{22} \right] + \left[Q_{12} - Q_{22} \right] - \frac{Q_{TX} \left[Q_{T2} + \lambda Q_{Tx} \right]}{Q_{TT}} = 0$$

from which it follows:

$$\lambda\left[Q_{11}-2Q_{12}+Q_{22}\right]Q_{TT}+\left[Q_{12}-Q_{22}\right]Q_{TT}-Q_{TX}\left[Q_{T2}+\lambda Q_{Tx}\right]=0$$

$$\lambda\ \left[\left[Q_{11}-2Q_{12}+Q_{22}\right]Q_{TT}-Q_{TX}^2\right]\ =Q_{TT}\left[Q_{22}-Q_{12}\right]+Q_{TX}Q_{T2}$$

and finally that:

$$\lambda\ =\ \frac{Q_{TT}\left[Q_{22}-Q_{12}\right]\ \ +Q_{TX}Q_{T2}}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]\ \ -Q_{TX}^2} \tag{4.32}$$

It remains to solve for τ from Equation 4.22 or 4.31. The latter implies:

$$\tau=-\frac{1}{Q_{TT}}\left[\frac{Q_{T2}Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{T2}Q_{TX}^2+Q_{TX}Q_{TT}\ \ \left[Q_{22}-Q_{12}\right]+Q_{T2}Q_{TX}^2}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right]$$

$$=-\left[\frac{Q_{T2}\left[Q_{11}-2Q_{12}+Q_{22}\right]+Q_{TX}\ \ \left[Q_{22}-Q_{12}\right]}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right]$$

$$=-\left[\frac{Q_{T2}Q_{11}+Q_{T2}Q_{22}-2Q_{T2}Q_{12}+\left[Q_{T1}-Q_{T2}\right]\ \ \left[Q_{22}-Q_{12}\right]}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right]$$

$$=-\left[\frac{Q_{T2}Q_{11}\ +Q_{T2}Q_{22}-2Q_{T2}Q_{12}+Q_{T1}Q_{22}-Q_{T1}Q_{12}-Q_{T2}Q_{22}+Q_{T2}Q_{12}}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right]$$

$$=-\left[\frac{Q_{T2}\left[Q_{11}+Q_{22}-2Q_{12}-Q_{22}+Q_{12}\right]+Q_{T1}\left[Q_{22}-Q_{12}\right]}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right]$$

Thus, finally we see that:

$$\tau=-\left[\frac{Q_{T2}\left[Q_{11}-Q_{12}\right]+Q_{T1}\ \ \left[Q_{22}-Q_{12}\right]}{Q_{TT}\left[Q_{11}-2Q_{12}+Q_{22}\right]-Q_{TX}^2}\right] \tag{4.33}$$

In the next section we describe how edges are represented and stored in our LoD system.

4.6 Implementation issues

We use a data structure similar to a hash map for storing and accessing edges. The edges of a model are stored in an EdgeArray (illustrated in Figure 4.12). To ensure rapid access, the edgearray has the same number of elements as there are vertices in the model and each element of the EdgeArray will have a pointer to a list of edge pairs. Edges are stored once in the EdgeArray with the smallest vertex id of the vertex pair defining an edge chosen as the first vertex of the edge. For any given edge query or look-up, two vertex ids are required. These are promptly sorted and the smallest id is used to look-up the EdgeArray position that contains the vertex pair.

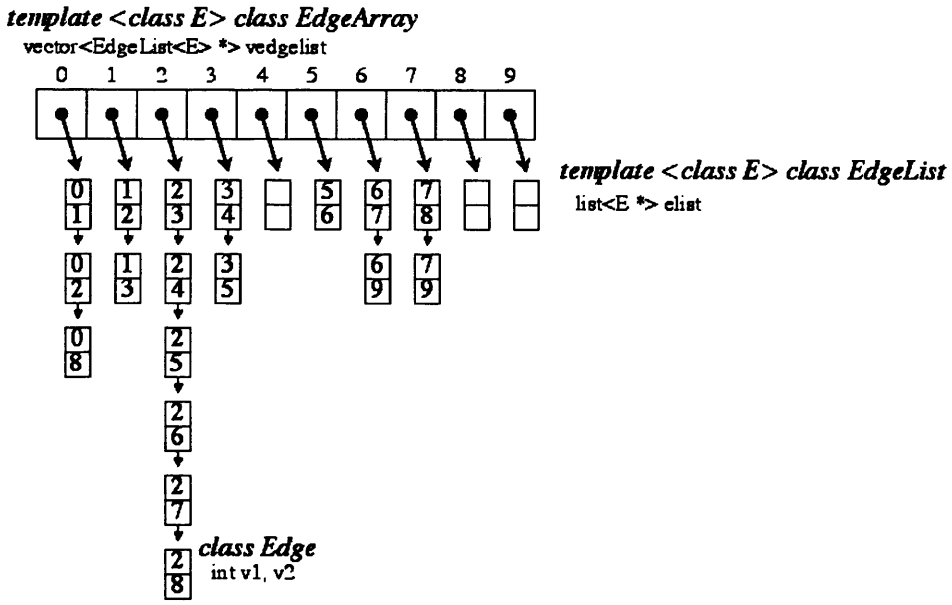


Figure 4.12: Edgearray data structure.

4.7 Results

In this section we use the method of automatically calibrating the solver presented in Section 4.3 directly to compare the geometric error obtained when we store and acummulate quadrics as in QSlim with that obtained when we do not store the quadrics in this way but calculate quadrics afresh based on the current mesh, with no history of previous edge collapses as in the memoryless approach of [LT99]. The effectiveness of the tolerance chosen by our calibration system is evaluated by comparing the geometric errors obtained when using it with a vertex placement strategy that always defaults to the edge midpoint location. We default the vertex to the edge midpoint as that is the position chosen in other competing strategies when the system of equa-

tions defining the optimal vertex location is ill-conditioned. It can be seen from the mean error results in Sections 4.7.1 and 4.7.2 that the condition tolerance chosen by the system enables our implementation of QSlim and the memoryless quadrics approach to produce better error meshes than a poorly chosen tolerance that does not take into account the units in which, or scale at which, the model is defined. Additionally it can also be seen from the figure in Section 4.7.3 that the approach in which we do not store the quadrics does indeed produce high quality meshes, comparable to or better than those of the QSlim method that does store and accumulates them as described above. The memory savings and the associated good quality of the meshes make such an approach well suited for simplifying large models such as the 1.7 million triangle turbine blade model. Storing the 882,954 quadrics comprised of 16 doubles would mean that a computer with 1 Gigabyte of RAM would run out of main memory, essentially owing to the memory footprint of 113 MBytes associated with the quadrics.

Deleting 1,087,716 triangles of the Buddha statue took 507 seconds with the memoryless quadric approach using a PowerPC G4 500 MGhz and took 447.9 seconds with our implementation of QSlim. The numerical comparisons between the two methods used the symmetric, dual pass error measuring tool Metro [Cignoni et al. [CRS98]], with the scan conversion sampling of edges, vertices and triangles.

4.7.1 Ciara body scan

Figure 4.13 presents the geometric errors obtained from Metro for uniform reduction of the Ciara body scan model using the memoryless quadrics approach, our implementation of QSlim and a midpoint placement strategy using memoryless quadrics. It can be seen that the strategy of midpoint placement has the highest error values of the three simplification methods. The figure shows that the threshold chosen by our calibration algorithm enabled the QSlim and Lindstrom's approach of not storing quadrics to obtain better geometric errors than the mid-point placement strategy by finding more optimal positions than mid position placements. Finally the figure also shows that it was possible to obtain geometric errors with Lindstrom's memoryless quadrics approach that were comparable with the QSlim approach.

4.7.2 Face scan

Figure 4.14 presents the geometric errors obtained from Metro for uniform reduction of the face scan model using three simplification approaches: the memoryless quadrics approach, our implementation of QSlim and a midpoint placement strategy using memoryless quadrics. In common with the results from Section 4.7.1, the automatically chosen condition number threshold enabled QSlim and the memoryless quadrics approach to achieve lower mean errors

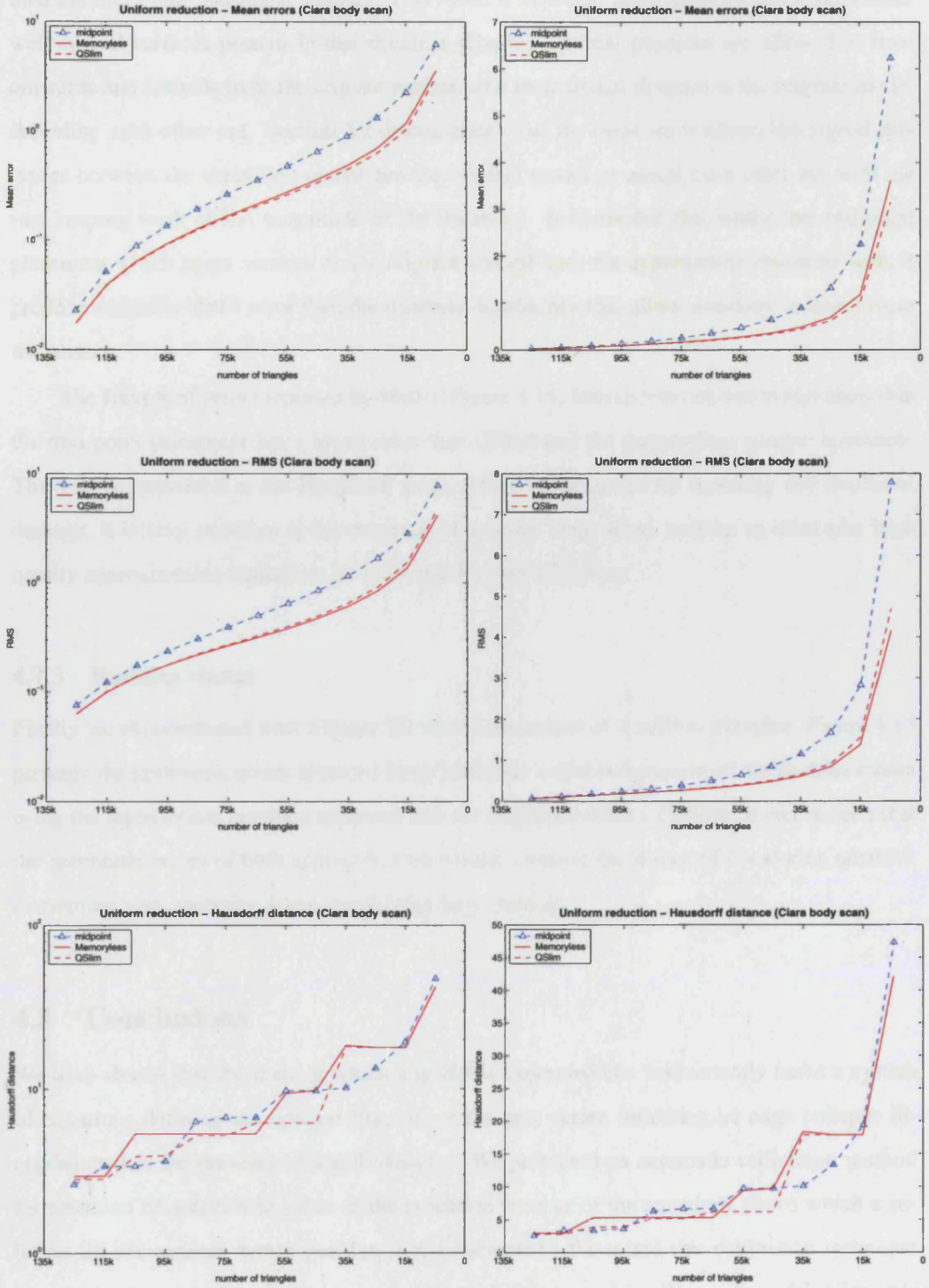


Figure 4.13: Memoryless quadrics simplification, QSlim simplification and midpoint placement strategy using memoryless quadrics for the Ciara body model. All techniques used the same calibrated solver, *top to bottom:* mean, RMS and Hausdorff distance, left, log scale plot of errors and right, linear scale plot of errors.

than the mid-point placement strategy. This result is expected since quadrics can approximate well curved surfaces present in this situation [Gar99], optimal positions are allowed to float outwards and inwards from the original surface with their signed distance to the original model canceling each other out. Section 2.2 demonstrates that the mean error allows the signed distances between the simplified model and the original model to cancel each other out with the rms keeping track of the magnitude of the distances. It is notable that whilst the mid-point placement which keeps vertices on the original surface does not approximate curvature well, it produce a smaller RMS error than the quadrics approaches that allow solutions to divert from the surface.

The Hausdorff errors reported by Metro (Figure 4.14, bottom row) on this model show that the mid-point placement has a lower error than QSlim and the memoryless quadric approach. This can be explained as the Hausdorff error, whilst being useful for reporting any structural damage, it is very sensitive to the presence of a single large error, making an otherwise high quality approximation equivalant to a low quality approximation.

4.7.3 Buddha statue

Finally we experimented with a larger 3D model comprised of 1 million triangles. Figure 4.15 presents the geometric errors obtained from Metro for uniform reduction of the Buddha model using the memoryless quadrics approach and our implementation of QSlim. It can be seen that the geometric errors of both approaches are similar, making the choice of not storing quadrics in memory very attractive when simplifying large models.

4.8 Conclusions

We have shown that the units in which a model is expressed can inadvertently make a system of equations defining the optimal location of the new vertex following an edge collapse ill-conditioned in the presence of small triangles. We presented an automatic calibration method for selection of a threshold value of the condition number of the equations above which a solution for the optimal vertex position is not attempted. We tested this calibration technique by carrying out a LoD reduction on a variety of different models. We confirmed Lindstrom's findings that an approach which does not store and accumulate the quadrics can create high quality meshes comparable to or better than those obtained with the approach adopted in QSlim in which quadrics are stored and accumulated. We presented a simple strategy for incorporating mesh quality constraints in LoD systems based on edge collapse. We derived the algebra

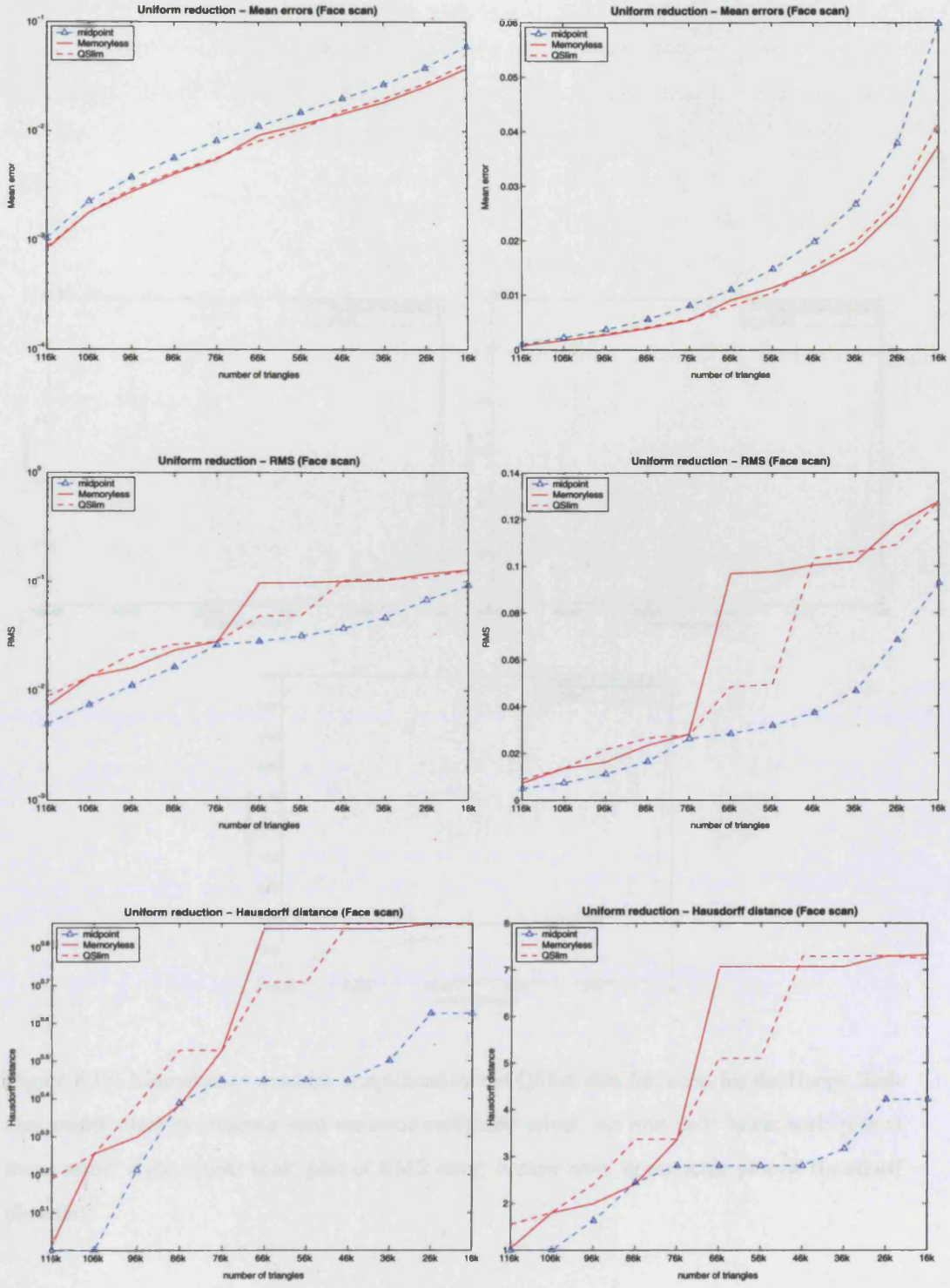


Figure 4.14: Memoryless quadric simplification, QSlim simplification and midpoint placement strategy using memoryless quadrics for the face scan model. All techniques used the same calibrated solver, *top to bottom:* mean, RMS and Hausdorff distance, left, log scale plot of errors and right, linear scale plot of errors.

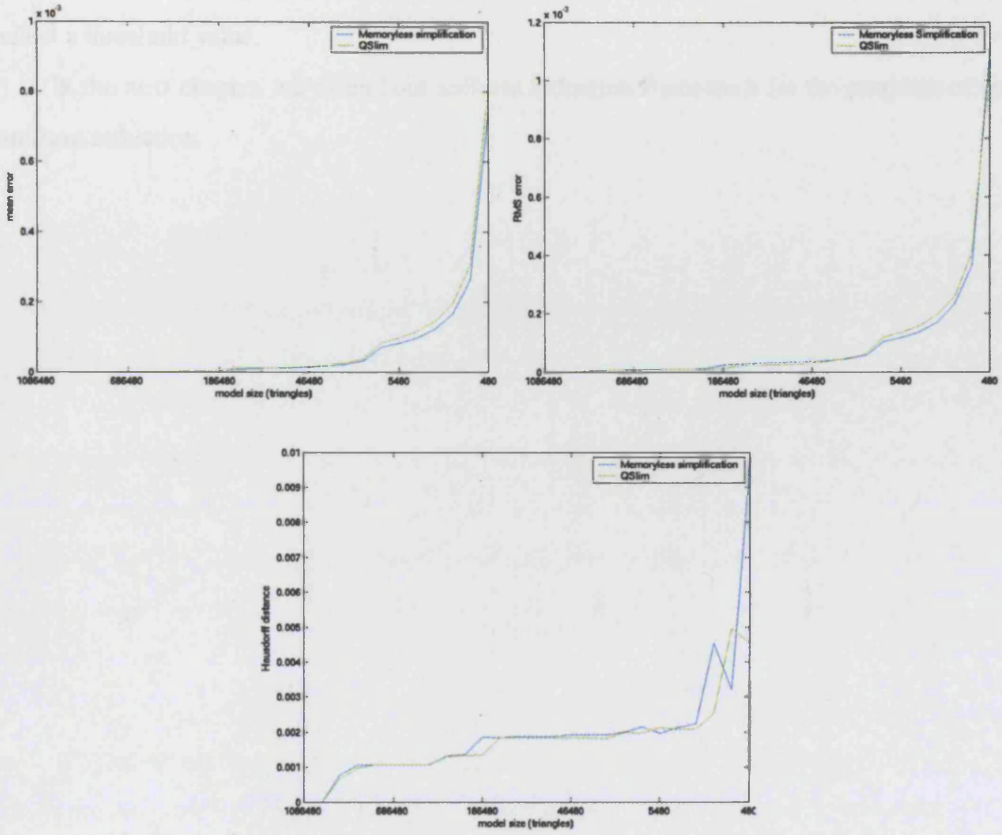


Figure 4.15: Memoryless quadrics simplification and QSlim simplification for the Happy Buddha model. Both techniques used the same calibrated solver, *top row:* left: linear scale plot of mean error; right: linear scale plot of RMS error; *bottom row:* linear scale plot of Hausdorff distance.

required for constraining a solution to lie on a 'perpendicular plane' in the treatment of border edges, thereby enabling us to use open surfaces in our work. Whilst the condition number calibration worked effectively with every model we used, we recognize that some models, for example, that may be completely planar or almost completely planar, might nevertheless produce a pathological result. In this situation, our automatic solution could be used to provide an initial estimate of the condition number threshold required. Our implementation also allows the user to visualise a graph of the condition numbers in rank order and, if desired, manually to select a threshold value.

In the next chapter, we extend our uniform reduction framework for the purposes of non-uniform reduction.

Chapter 5

Vertex classification for non-uniform geometry reduction

5.1 Introduction

We can now interact and mark features in a complex model and have a simple strategy for easily incorporating mesh quality constraints into an LoD system based on edge collapses using measures such as the quadric error. In this chapter we extend our strategy to preserve marked features and simplify non-uniformly semi-regular models. We study the simplification errors of our non-uniform system using three different edge collapse algorithms: Our implementation of QSlim that stores and accumulates quadrics, the memoryless approach that recalculates quadrics when needed and doesn't store them and a mid-point vertex placement strategy using the memoryless approach.

In the context of the quadric error studied in the previous chapter we considered the problem of determining the condition number threshold for matrix inversion and presented an automatic solution. We confirmed Lindstom's findings that recalculating the quadrics still produces high quality meshes when compared with a condition threshold that defaults only to the mid-point placement strategy and with QSlim's strategy that stores and accumulates the quadrics. We now examine the errors of the same simplification strategies in the non-uniform reduction setting with the same 3D models. We draw conclusions from comparing results from uniform and non-uniform reduction in Chapter 7.

In Section 5.2 we extend and add further constraints to the uniform reduction framework presented in Section 4.4.4 to handle non-uniform reduction of semi-regular meshes. We present examples of different types of non-uniform reduction, such as that using vertex classification and preservation of manually selected feature areas in Section 5.2, vertex classification and preservation of automatically set regions around joints in the context of an animation system in Section 5.2.2, absolute preservation of border features in Section 5.3 and the preservation of

colour discontinuities in Section 5.4. We study their respective simplification errors at the end of each section. Finally in Section 5.5 we present conclusions.

5.2 Non-uniform geometry reduction - System Heuristics

In this section we present our framework for non-uniform reduction of semi-regular meshes. The first step is to mark buffer regions around the vertices we wish to preserve. Edges in user selected regions have a large *offset* cost added to their geometric cost to prevent them from being simplified. At any time the cost of these edges can be recalculated and the offset cost removed to encourage simplification in those areas. Edges can only be collapsed with buffer vertices from the same region. These buffers use triangle aspect ratio constraints to handle the transition between the fine detail triangles we wish to preserve and the coarse triangles. Edges from buffer regions do not have an offset cost added. Algorithm 4 shows how a number of buffer regions *nbuffers* can be marked automatically around the feature vertices that are user selected and marked with a tag value (pA->fv) of 1. Unmarked vertices will have a tag value of zero and no buffers are created around them.

Algorithm 4 Pseudo-code for buffer vertex classification.

```
int function create_buffers (int nbuffers)
// where nbuffers >=1
// tag unmarked vertices immediately connected to a k type vertex with a new buffer id
for(k=1 k<=nbuffers k++)
{
    for(vid=0 vid<number_of_vertices vid++)
    {
      pA=atVertexArray(vid)
      if(pA->fv==k)
      {
        for(face ∈ pA->flist)
        {
            for(pB ∈ face)
            {
              if(pB->fv==0) pB->set_fv(k+1)
            }
        }
      }
    }
}
return 0
```

Figure 5.1 shows two buffer regions in yellow and green around manually marked features in dark blue. The figure also shows the automatic condition number threshold by means of the red line. In the far right of the graph we see that only a few potential edge collapses are ill-conditioned.

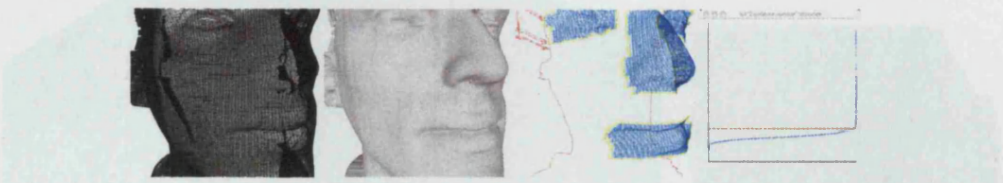


Figure 5.1: Manually marked features of the face scan model for non-uniform reduction - from left to right: wireframe rendering of original model [126,108 triangles]; Gouraud shading of original model; manually selected features in dark blue, 2 interface buffers in yellow and green [5,433 selected triangles, 1,946 borders out of a total of 190,135 edges]; condition number calibration [*wthres*=2.1x10^6; *wmax*=2.6x10^26; *wmin*=195.6]

As with our mesh quality constraints presented in the previous chapter we do not make explicit changes to the simplification algorithms used but use an objective constraint approach to ensure that different algorithms can be used with our vertex classification system. In order to manage the interface between fine detail and coarse larger triangles we observe that in a semi-regular uniform mesh a vertex typically has a fixed number of edge connections to other vertices. This number increases as the mesh becomes coarser on one side. A simple edge valence constraint *does_overedges* added to the decimation loop (see Section 4.4.4) can signal when potentially too many connections are made and prevent and penalize the edge collapse. The typical edge valence of vertices of the semi-regular meshes used in this thesis is six (as shown in the two laser scans in Figure 5.2). We found that constraining the edge valence to nine or less prevents excessive branching. Another property of a non-uniformly reduced mesh in these interface areas (regardless of which simplification algorithm is used) is that the aspect ratio of connecting triangles changes. Our system allows the cost function to prioritize edge collapses but cascades the required changes of triangle aspect ratio over different buffers.

Figure 5.2 shows the triangle aspect ratio using Gueziec's [Gue96] formula for randomly selected triangles in two semi-regular scanned meshes. A value of 1.0 would indicate an equi-lateral triangle, and a value of zero a degenerate thin triangle. On the right of the face scan, triangles with a low aspect ratio can be seen in the region of the ear.

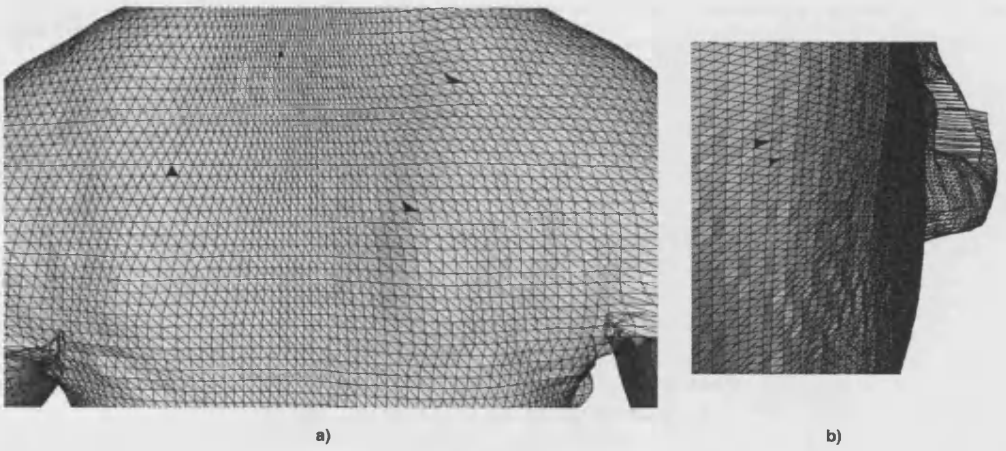


Figure 5.2: Triangle aspect ratio [Gue96] of sampled triangles in black from semi-regular scanned meshes [values of triangle aspect ratios from left to right]- a) Ciara body scan:0.97, 0.6123, 0.658, 0.6122;b)face scan:0.704, 0.866, 0.092 (long thin triangle located in the right ear).

We changed the triangle aspect ratio constraint *does_mesh_thin* presented in the previous chapter (Section 4.4.2) to handle two types of triangle aspect ratio constraints:

- One, as before, to signal the creation of a thin triangle anywhere in the mesh. We use as reference the tolerance of the long thin triangles on the right image of Figure 5.2 (e.g. 0.092) to set a conservative tolerance *tasp* of a value of 0.16. We note that the original triangles in general in a regular mesh have an aspect ratio value of 0.6 or greater.

- We use a second tolerance *taspb* specifically for triangles formed from buffer vertices and signal that a triangle has reached a distortion limit if its triangle aspect ratio is lower than a value of 0.4.

Algorithm 5 outlines the changes made for the detection of thin triangles. If an *offset* of zero is selected, the algorithm performs uniform reduction.

Algorithm 5 Pseudo-code for thin triangle detection.

```
int function does_mesh_thin (collapse_struct c)
P1=atVertexArray(c->edge->vid1)
P2=atVertexArray(c->edge->vid2)
for(face ∈ p1->flist)
    //code block A
    if(face->mark!=2)
        casp=compute_aspect_ratio(face) //with P1's position
        asp=compute_aspect_ratio(face) //with P1's after collap. pos.
        if(casp<tasp)
            return 0 //feature or not, tri. was already bad, allow collapse
        else
            if(offset>0) //LOD solver is in non-uniform mode
              if((P1->feature>0)\/(P2->feature>0)...
              \/(P3->feature>0))... //feature area
                  if(asp<tasb) //apply buffer area tolerance
                      return 1
              else //non feature area
                  if(tasp>0) // user chose to avoid thin triangles
                      if(asp<tasp) //apply uniform area tol.
                          return 1
for(face ∈ p2->flist)
... //code block A
return 0
```

Figure 5.3 and Figure 5.4 show the results of adding one at a time each of the three constraints to the decimation loop of our implementation of QSlim as designated from left to right. The features used for preservation are shown in Figure 5.1.

The first leftmost column of Figure 5.3 shows only the *does_overedges* constraint being used. A rendering artefact is visible in the corresponding first leftmost column of Figure 5.4 towards the left of the upper lip where as the mesh cuts across the lip it folds itself. This artefact disappears when one adds the *does_mesh_foldover* constraint (second column). Finally, the rightmost column shows how the modified *does_thin_triangle* constraint improves the triangle aspect ratio distribution over the different buffers in the area near the nose.

Finally Algorithm 6 shows how the different constraints have been added in the decimation loop.

In the following subsections we present the geometric error performance of different simplification functions using our non-uniform framework with a face scan in Section 5.2.1 and a body scan in Section 5.2.2.

5.2.1 Results - Face scan

It took 27 seconds to reduce non-uniformly with the three constraints the 126,108 triangles face scan model to the 10,000 triangle LoDs shown in Figure 5.3 and Figure 5.4 with a G4 500

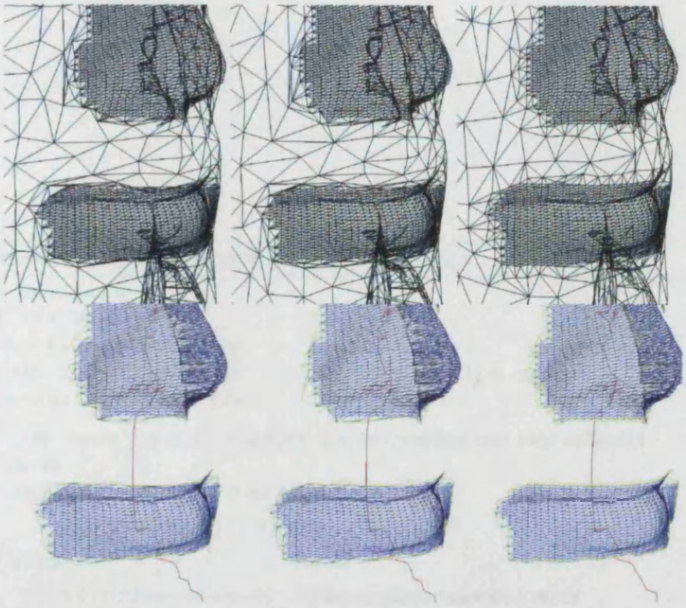


Figure 5.3: Wireframe rendering of non-uniform reduction constraints with the face scan model and QSlim simplification. The original model has 126,108 triangles, the LoDs have 10,000 triangles, the preserved triangles in dark blue account for 5,433 triangles, the triangles in two yellow and green buffers account for 1,900 triangles, and there are 2,667 regular triangles - *left:* *does_overedges* *middle:* *does_overedges* + *does_mesh_foldover* *right:* *does_overedges* + *does_mesh_foldover* + *does_thin_triangle*.

Figure 5.4: Gouraud shaded rendering of LoD models (shown in Figure 5.3) with non-uniform reduction constraints applied to the face scan model and QSlim simplification used. The original model has 126,108 triangles, the three LoDs shown have 10,000 triangles: *left:* with the *does_overedges* constraint (a dark rendering artefact is visible in the centre of the upper lip where the mesh folded over), *middle:* with two constraints designated by: *does_overedges* + *does_mesh_foldover* (the mesh fold over no longer occurs) *right:* with three constraints designated by: *does_overedges* + *does_mesh_foldover* + *does_thin_triangle* (thin triangles which can also create rendering artefacts are prevented, not visible here, but see Figure 6.24 and Figure 6.25-a).

Algorithm 6 Pseudo-code for Non-uniform LoD heuristics

```
function decimate ()
    deadlock = 0
    deadlocklimit = #edges
    #target_triangles = #triangles-decimation_rate
    while(#triangles > #target_triangles)
        c=h->get_heap[0]// get top of heap (smallest cost edge collapse)
        go=1
        if(#edges == 0) // no edges left
            print "done!" break
        else
            if(offsetcost>0) // Non-uniform reduction mode
                if(P1->feature==P2->feature)
                else go=0 // vertices have to be of the same type

                if(go)          // edge valence test
                    if(does_overedges(c)) go=0
                if(go)          // mesh foldover?
                    if(does_mesh_foldover(c)) go=0
                if(go)          // thin triangle generated?
                    if(does_mesh_thin(c)) go=0
            else // Uniform reduction mode
                if(does_mesh_foldover(c))//mesh foldover?
                    go=0
                if(go)
                    if(does_mesh_thin(c)) // thin triangle-
                      go=0                // -generated?
            if(go)
                deadlock=0
                do_edge_collapse(c)
            else
                if(deadlock==deadlocklimit)// deadlock-
                  deadlock=0 // -limit reached, allow progress
                  do_edge_collapse(c)
                else
                  deadlock=deadlock+1
                  c2=h->get_heap[#current_edges-1]
                  h->re_insert(c,c2.cost+1000)
```

MGhz PowerPC desktop, versus 18 seconds if it was reduced uniformly with the same QSlim implementation and no mesh quality constraints.

We simplified the model using our vertex classification system with three different simplification algorithms:

- Our implementation of QSlim that stores and accumulates quadrics.
- The memoryless approach that recalculates the quadrics instead of storing them.
- A memoryless mid-point placement strategy.

To avoid the borders of the model receding, the border vertices were classified and the border quadric from Section 4.5 used with each simplification algorithm.

Figure 5.5 shows the geometric errors reported by Metro with its symmetric scan option. It shows that the memoryless approach yields a lower mean error than that obtained when storing and accumulating the quadrics during non-uniform reduction.

As expected, the memoryless midpoint vertex placement strategy that confines the solutions to lie on the surface yields the largest mean error results. As with uniform reduction, the RMS error and Hausdorff distance for this open surface model report a lower error for the midpoint placement strategy (see Section 4.7.2 for an explanation of this result).

Although this is an extreme case of reduction where the number of preserved triangles accounted for half the triangles of the LoD, the model can be further reduced if the user resets the offset cost of selected edges to zero and the system recomputes the edge costs.

5.2.2 Results - Ciara body scan (vertex classification for higher quality soft deformation animation)

A whole-body scan produces models that were too complex for real-time animation on our platform. The number of triangles to render and the computational demands of applying matrix transformations and matrix blending to a large number of vertices are both prohibitively high.

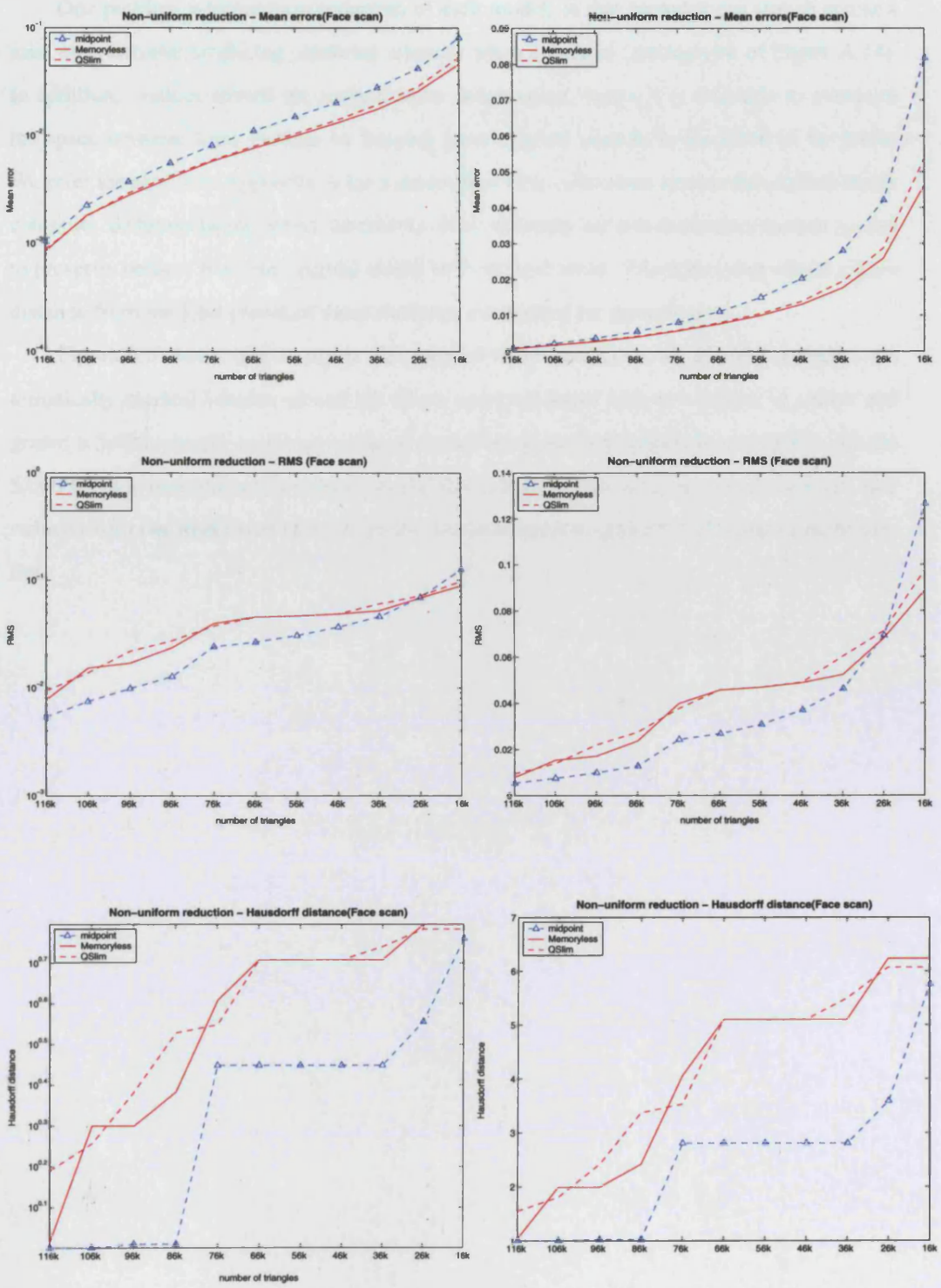


Figure 5.5: Non-uniform reduction, feature preservation in manually marked face regions - geometric errors for the face scan model, *top to bottom:* mean, RMS and Hausdorff distance, left, log scale plot of errors and right, linear scale plot of errors.

One problem with uniform reduction of such models is that triangles can stretch across a joint area, thereby producing rendering artefacts when animated (see top row of Figure A.14). In addition, vertices stretch the surface under deformation, hence it is desirable to minimize the space between these vertices by keeping more original vertices in the areas of the joints. We refer the reader to Appendix A for a description of an animation system that automatically computes skeletons using surface landmarks. Here we apply our non-uniform reduction system to preserve vertices from the original model in these joint areas. Triangles lying within a fixed distance from the joint planes of these skeletons are marked for preservation.

Figure 5.6 shows (left to right): the original body scan model of 135,192 triangles, automatically marked features around the elbow and knee joints with two buffers in yellow and green; a 5,000 triangle uniformly reduced model using our implementation of QSlim and the 5,000 triangle non-uniformly reduced model that keeps original vertices around the joints also reduced with our implementation of QSlim. Uniform and non-uniform LoDs appear quite similar.

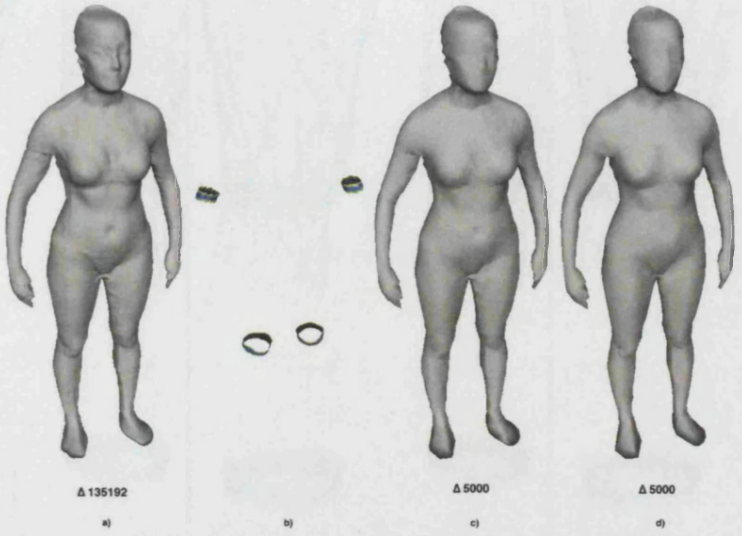


Figure 5.6: Automatically marked features around joint areas of the Ciara body scan model for non-uniform reduction - from left to right: Gouraud shaded rendering of original model [135,192 triangles]; marked features in dark blue and 2 interface buffers in yellow and green; uniformly reduced LoD of 5,000 triangles with QSlim simplification; non-uniformly reduced LoD of 5,000 triangles with QSlim simplification.

Figure 5.7 shows a close-up of the knee joint area.

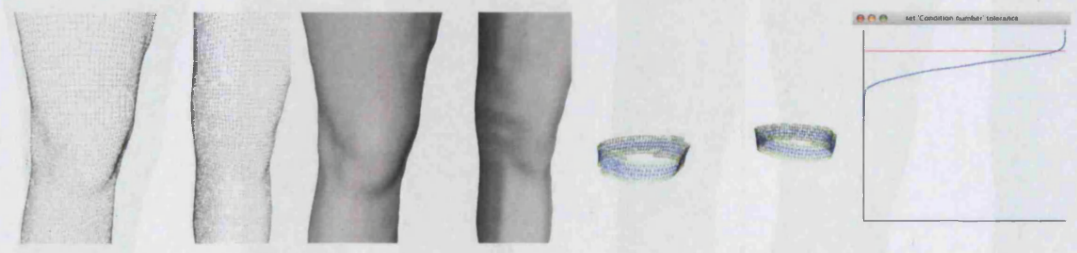


Figure 5.7: Close-up of automatically marked features around knee joint areas of the Ciara body scan model for non-uniform reduction - from left to right: wireframe rendering of original model [135,192 triangles]; Gouraud shading of original model; marked features in dark blue and 2 interface buffers in yellow and green; condition number calibration [*wthres*=5.8x10^11; *wmax*=1.4x10^13; *wmin*=7.9x10^6].

Figure 5.8 and Figure 5.9 show a closeup of the results of the three different simplification strategies using our vertex classification system.

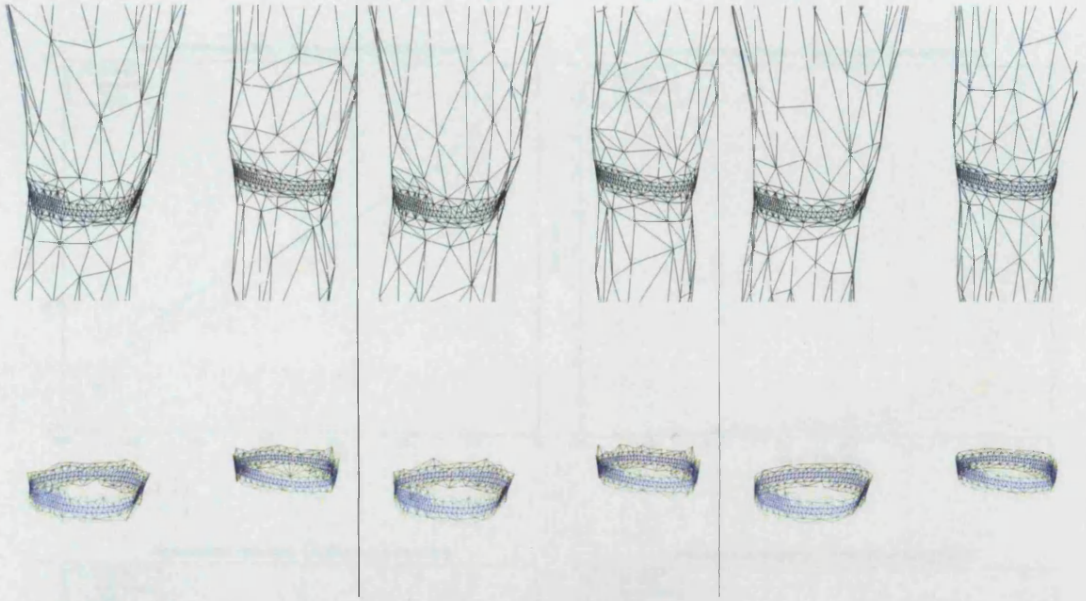


Figure 5.8: Close-up of wireframe rendering of 5,000 triangle non-uniformly reduced LoDs using three different simplification strategies: *left:* memoryless simplification; *middle:* QSlim; *right:* midpoint placement with memoryless quadrics.

Figure 5.9: Close-up of Gouraud rendering of 5,000 triangle non-uniformly reduced LoDs using three different simplification strategies: *left:* memoryless simplification; *middle:* QSlim; *right:* midpoint placement with memoryless quadrics.

Figure 5.10 shows that as with the face scan model, the memoryless approach yields smaller mean errors.

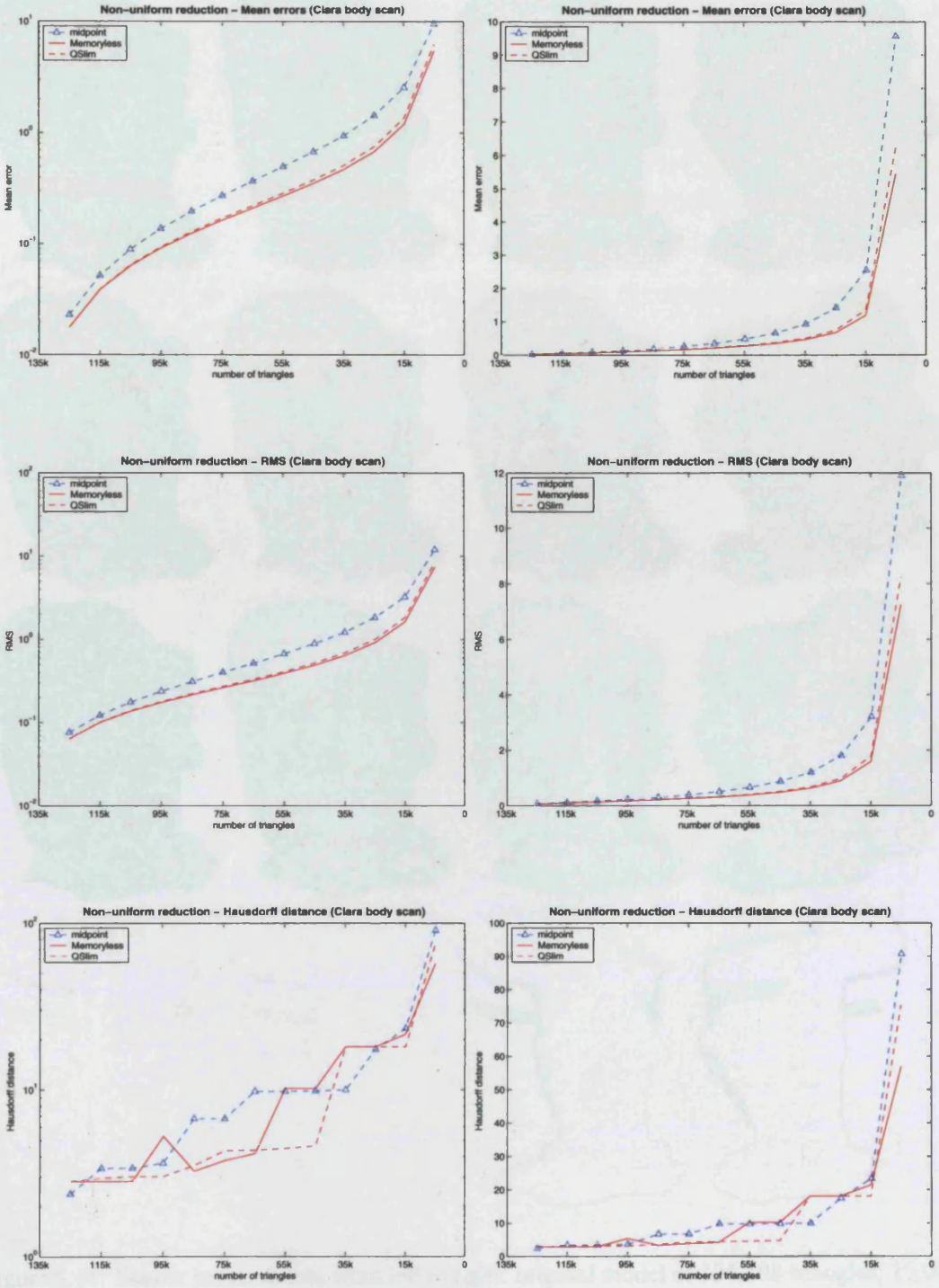


Figure 5.10: Non-uniform reduction for preservation of automatically marked features around joint areas of the Ciara body scan model - geometric errors for the Ciara body scan model, *top to bottom:* mean, RMS and Hausdorff distance, left, log scale plot of errors and right, linear scale plot of errors.

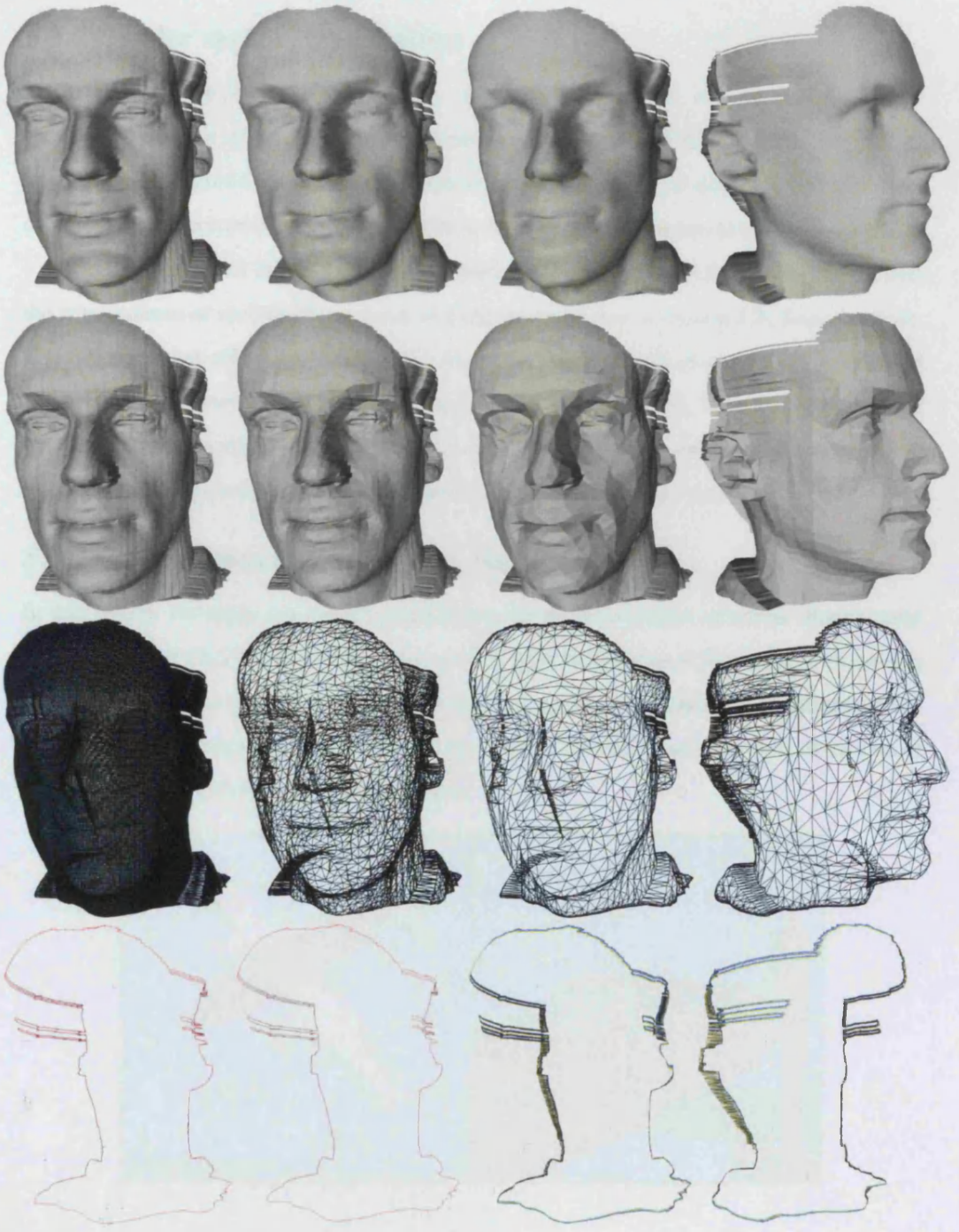


Figure 5.11: Border preservation, from left to right: original model of 126,108 triangles; 10,000 triangles LoD representations obtained using QSlim with border preservation; two right most images: 10,000 triangles LoD representations obtained using memoryless quadrics and 2 buffer regions.

5.3 Border vertex classification

3D models of the human face play an increasingly important role in sugery, healthcare [KGC+96] and on games and entertainment. A simulation tool might require that vertices from the original model are kept for registration purposes. Hence it is desirable that a simplification algorithm can preserve original border vertices in the LoD representations produced.

One problem with Garland's method of preserving borders is that there is no control over the edge valence of such border vertices, making the mesh over constrained in those locations. The two rightmost columns of Figure 5.11 show our vertex classification system that has edge valence constraints with memoryless quadrics and two buffer regions. More triangles are used than in results obtained from than QSlim around border vertices to prevent excessive ramification of edges that would compromise the quality of the mesh at those locations.

5.4 Colour discontinuity vertex classification

In this section we apply our vertex classification for the preservation of colour discontinuity curves. We used the CAD cessna model shown in Figure 5.12 and used four ways of measuring the error: qualitative assessment of LoD renderings, qualitative assessment of difference images, quantitative assement of geometric errors reported by Metro and error plots of the sum of the absolute pixel difference of different views.

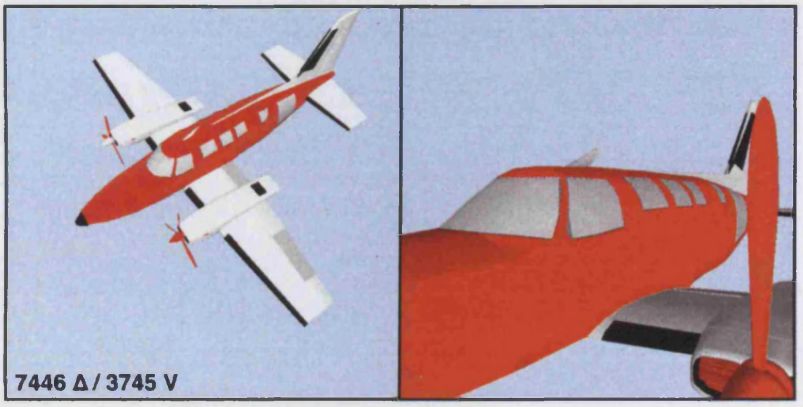


Figure 5.12: Colour discontinuity curves on the Cessna aeroplane CAD model comprised of 7,446 triangles.

Because the model is not a regular scan, we did not create buffer regions. Instead, we classified vertices along discontinuity curves and allowed edge collapses only with vertices of the same class. Figure 5.13 shows uniform geometry reduction without colour or mesh quality constraints in the left column. The middle column shows LoD representations obtained when

using the colour vertex classification test, the mesh fold-over test and the thin triangle test. The righthand column shows LoDs with the same constraints as in the middle column but without the thin triangle test being used. It can be seen that the mesh becomes over-constrained higher up in the middle column. This can also be seen in Figure 5.14.

Figure 5.13: Colour discontinuity preservation and geometric quality trade-off on Cessna aeroplane CAD model - *left:* no colour constraints (uniform reduction); *middle:* colour test+mesh fold-over test+thin triangle test [CMT]; *right:* colour test+mesh fold-over test [CM]. It can be seen that the unconstrained approach of the left column generates the best geometric results at the expense of colour discontinuity preservation. The middle column shows the effect of the system becoming overconstrained. On the right column with fewer constraints, it can be seen that even though the original model does not consist of many triangles a good compromise between geometric quality and colour discontinuity preservation can be achieved with 50% reduction.

Figure 5.14 shows the results of subtracting the rendering of the original model from the 2,446 triangle LoDs. A black image would indicate an identical rendering, bright pixels indicate areas where the pixels were different.

Figure 5.14: Image subtraction of 2,446 triangle LoD renderings and original model rendering. Pixels in black in the right three images indicate identical pixels from the LoD rendering and the original model rendering on the left, in constrast pixels in light blue indicate large differences. Left to right: original model; no colour constraints (uniform redution); colour test+mesh fold-over test+thin triangle test; colour test+mesh fold-over test.

Unlike the geometric error reported by Metro that is invariant from any view point, the perceived quality of a model can be different when seen from different viewpoints. Figure 5.15 shows close-up renderings of the same LoDs.

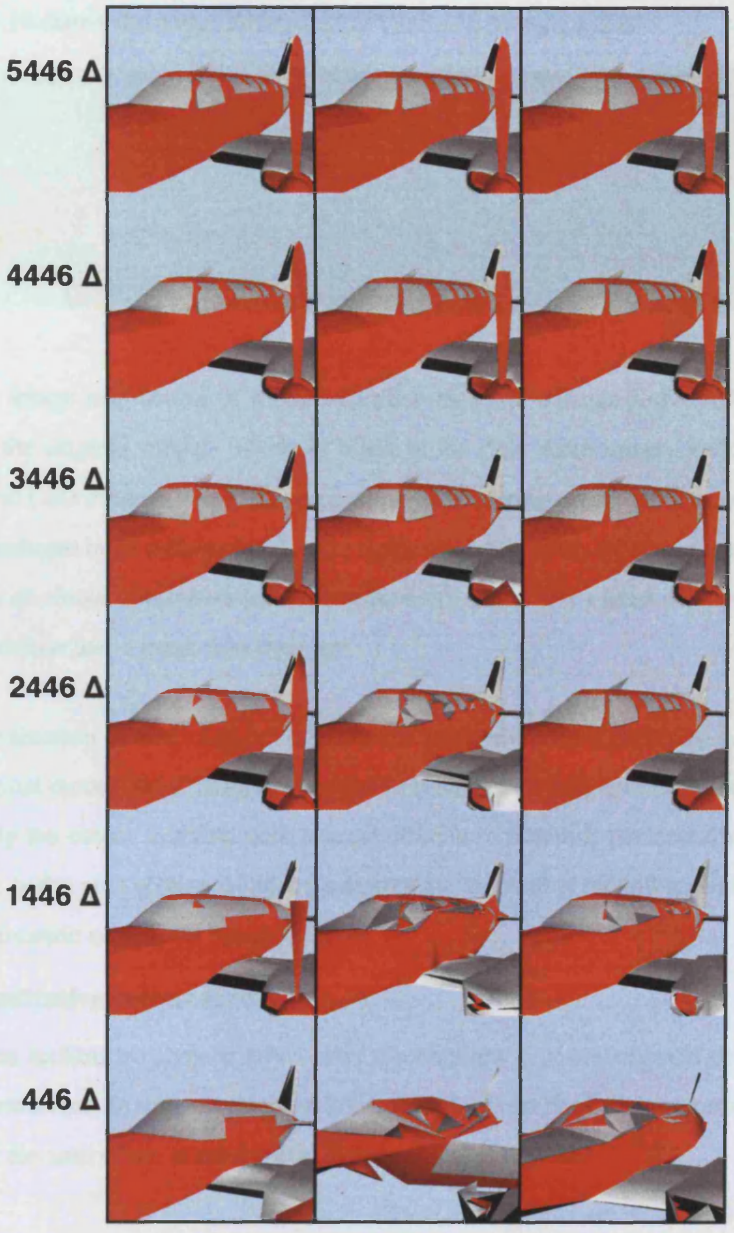


Figure 5.15: Close-up of colour discontinuity preservation on the Cessna aeroplane CAD model - *left:* no colour constraints (uniform reduction); *middle:* colour test + mesh fold-over test + thin triangle test [CMT]; *right:* colour test + mesh fold-over test [CM]. Again, it can be seen that the unconstrained approach of the left column generates the best geometric results at the expense of colour discontinuity preservation. The middle column shows the effect of the system becoming overconstrained. On the right column with fewer constraints, it can be seen that even though the original model does not consist of many triangles a good compromise between geometric quality and colour discontinuity preservation can be achieved with 50% reduction.

Figure 5.16 shows the image differences of the 3,446 triangle LoDs.

Figure 5.16: Image subtraction of a close-up view of 3,446 triangle LoD renderings and a rendering of the original model. Pixels in black in the right three images indicate identical pixels from the LoD rendering and the original model rendering on the left, in constrast pixels in light blue indicate large differences. Left to right: original model; difference images for LoD obtained with no colour constraints (uniform reduction); colour test + mesh fold-over test + thin triangle test; colour test + mesh fold-over test.

In our evaluation of how closely both uniform and non-uniform reduction LoDs can represent an original model, the differences or relative benefits between uniform reduction and the LoD with only the colour and fold-over test are difficult to establish perceptually. To address this difficulty, in the next section we plot the sum of the absolute pixel difference for each LoD and report geometric error from Metro.

5.4.1 Quantitative colour errors

In the previous section we showed how colour discontinuity constraints could be used to preserve colour attributes. In order to establish better the effectiveness of our constraints we plotted for each LoD the sum of the absolute pixel difference over all pixels:

$$pdif = fabs(R2 - R1) + fabs(G2 - G1) + fabs(B2 - B1) \tag{5.1}$$

where R1 and R2 stand for the red channel pixel components, G1 & G2 for the green and B1 & B2 for the blue.

Figure 5.17 shows the sum of the absolute pixel difference over all pixels for the two viewpoints used in the previous section. Figure 5.18 shows the geometric errors reported by Metro.

It can be seen from Figure 5.17 and Figure 5.18 that the LoDs that have best geometric error are not necessarily the LoDs that generate the smallest pixel error as defined by Equation 5.1 from the two different viewpoints and over the first 50% of reduction. Reducing the

7,446 triangle model using only the colour test and fold-over test preserves colour better than the other methods until there are as few as 2,446 triangles as seen from one view point, and until approximately as few as 3,446 triangles from the close-up viewpoint.

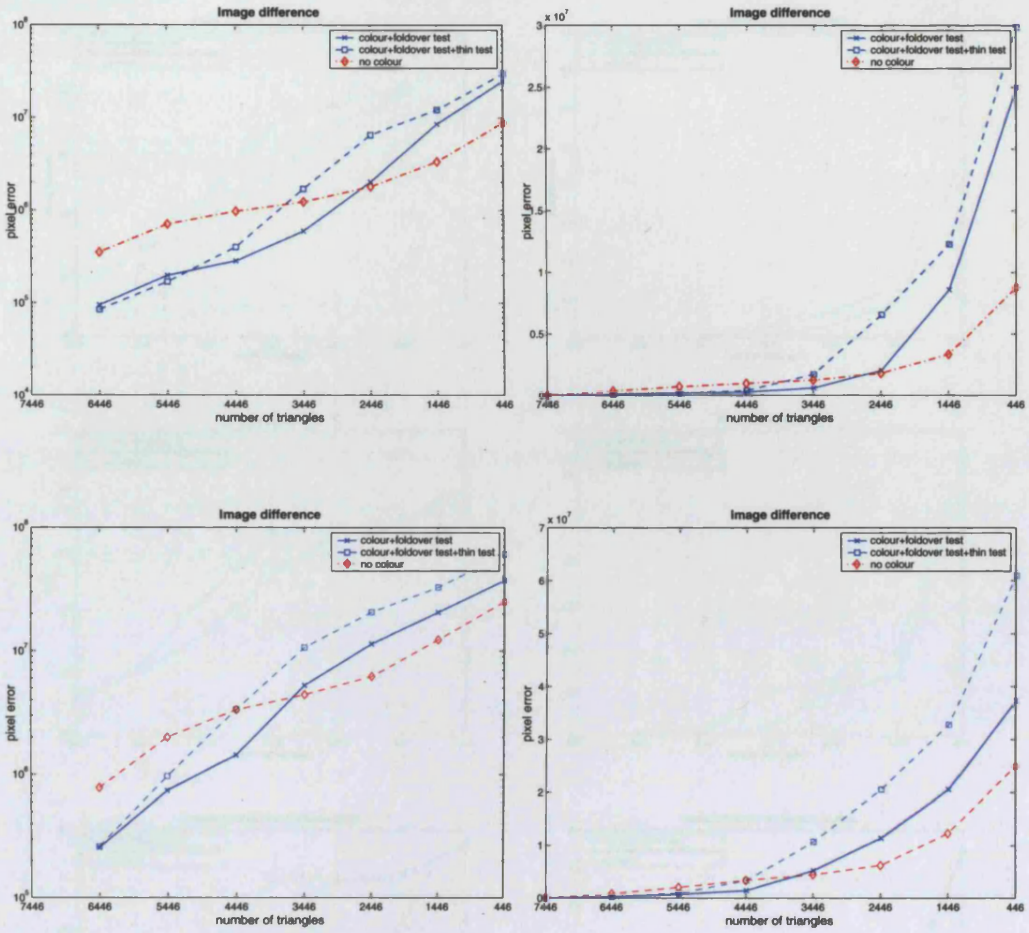


Figure 5.17: Sum of absolute pixel difference for each LOD of the Cessna aeroplane CAD model: *top:* full view image differences, *bottom:* close-up image differences; left, log scale plot of image differences and right, linear scale plot of image differences.

5.5 Conclusion

We presented a simple framework for reducing non-uniformly semi-regular models such as those from laser scans according to three different simplification algorithms. In particular, the memoryless quadrics approach produces geometric errors comparable to or better than those obtained with the QSlim approach of storing and accumulating the quadrics. We showed how our vertex classification strategy with buffer regions could preserve border edges and both man-

ually marked features and automatically detected features such as the joint areas of an animation model. Finally, we showed how the constraints could be relaxed to preserve colour discontinuity curves in a CAD model.

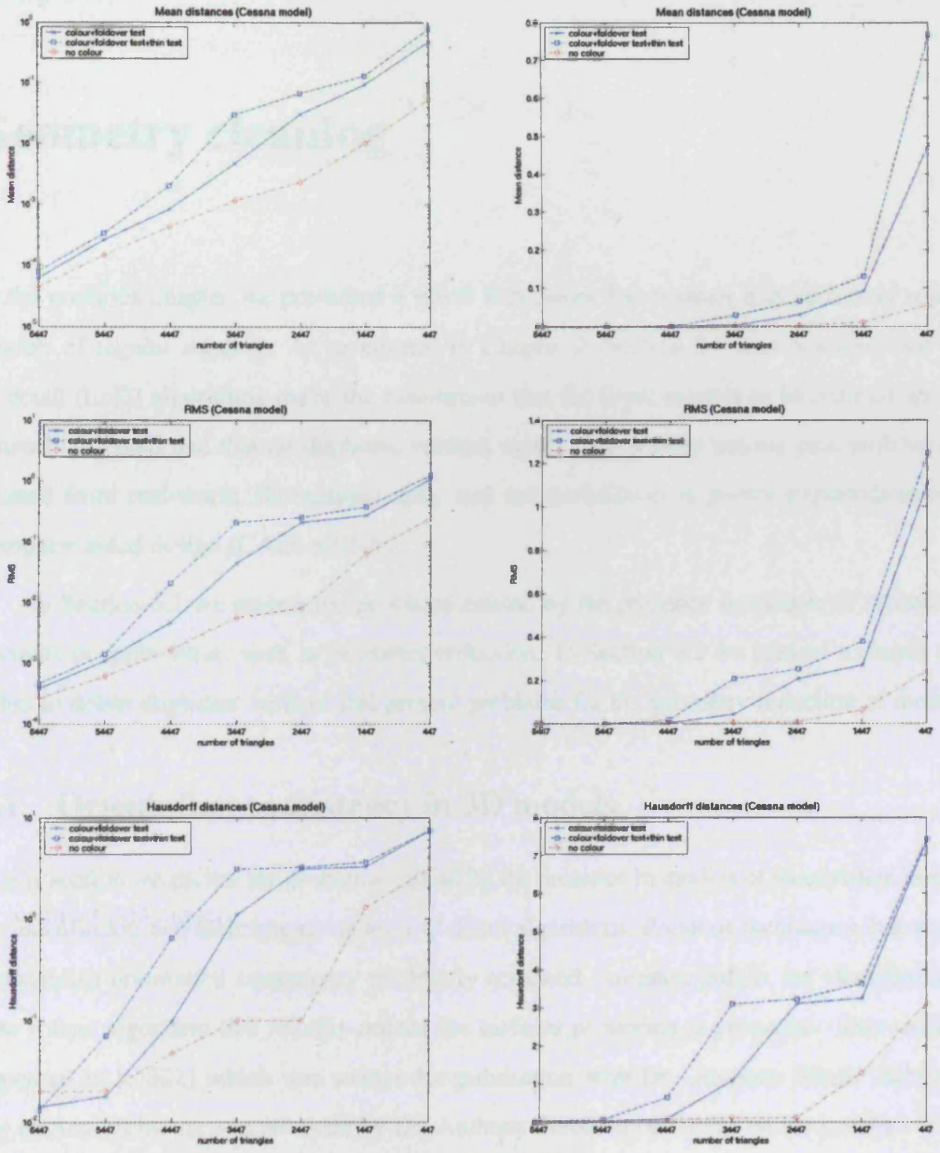


Figure 5.18: Non-uniform reduction for preservation of colour discontinuities - geometric errors for the Cessna aeroplane CAD model, *top to bottom:* mean, RMS and Hausdorff distance, left log scale plot of errors and right, linear scale plot of errors.

Chapter 6

Geometry cleaning

In the previous chapter we presented a novel framework for creating non-uniformly reduced models of regular meshes. As mentioned in Chapter 2, Section 2.1 it is common that level of detail (LoD) algorithms make the assumption that the input models to be reduced are consistently oriented and that no duplicate vertices exist. This is often not the case with surfaces created from real-world 3D scanned data, and not uncommon in poorly exported/converted computer aided design (CAD) models.

In Section 6.1 we review the problems caused by the presence in models of inconsistent normals in applications such as geometry reduction. In Section 6.2 we present a simple algorithm to delete duplicate vertices that present problems for the geometry reduction of models.

6.1 Orientation consistency in 3D models

In this section we review the problems caused by the presence in models of inconsistent normals in visualisation and their impact on level of detail algorithms. Previous techniques that attempt to establish orientation consistency are briefly reviewed, common pitfalls are identified and a new robust algorithm that reliably orients the surfaces of models is presented. This work has appeared in [OS02] which was written for publication with Dr. Anthony Steed. Advice on ray tracing problems was provided by Dr. Anthony Steed, in particular on the problem of rays intersecting surfaces at a tangential angle (Figure 6.8) which was instrumental in developing a reliable solution.

6.1.1 Normals in Computer Graphics, Visualisation

At a very basic level, vertex or triangle normals provide a means of evaluating the relationship between a surface and a light source, and the relationship between a surface and a viewer (Figure 6.1).

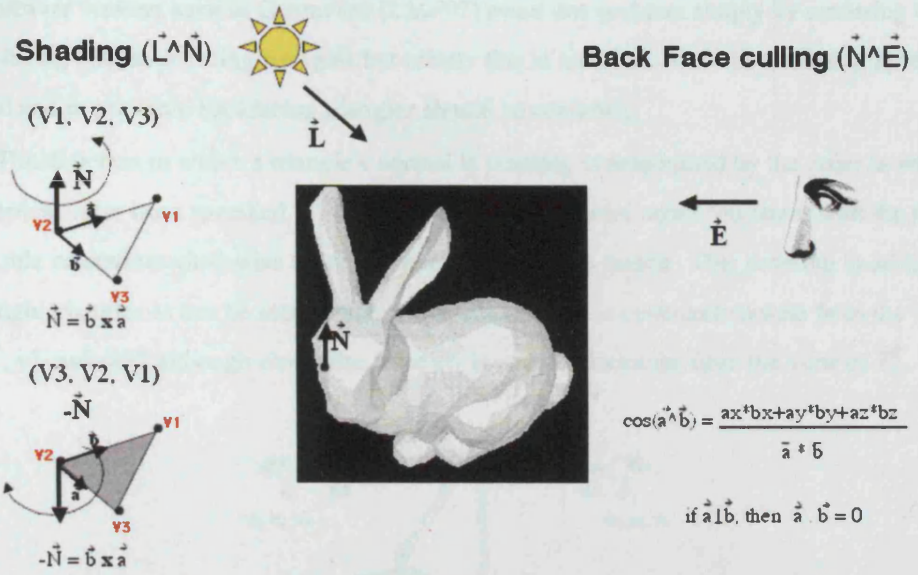


Figure 6.1: The relationship between surface normals and the direction of illumination and the view direction.

Vertex and triangle normals are used to determine the light intensity with which a particular part of a surface should be rendered. Surface attributes such as colour and texture can have their intensity changed according to the angle the normal makes with the direction of the light source. Surfaces facing the direction of illumination will be brightly lit whilst surfaces pointing away from the source will be darker. There are three main ways of polygon shading. In flat shading, one calculates the brightness of a triangle with one normal representing the entire triangle. This intensity value is then applied equally in a scan line fashion across each triangle. In Gouraud shading, or interpolated shading, we use the normals at each vertex of a triangle, and find three intensity values which are then interpolated across the triangle. Finally Phong shading interpolates normals from each vertex, normalizes them at each pixel and calculates a brightness for every pixel [FVFH90]. Another useful relationship that surface normals allow us to establish is visibility. Surface normals allow us to determine whether a surface is facing away from the viewer and hence whether the surface is visible or not. In computer graphics, one often renders only triangles that are facing the viewer to speed up the rendering. This process is referred to as *back face culling*.

In such systems, having inconsistent surface normals produces visually incorrect results. For example, Figure 6.30 *(leftmost* column) and Figure 6.31 *(rightmost* column) show white gaps in the original model where the triangle normals are pointing inwards to the model. Some

3D software viewers such as Geomview [LMP97] avoid this problem simply by rendering both front-facing and back-facing triangles but clearly this is not ideal since the rendering speed is halved and in principle backfacing triangles should be occluded.

The direction in which a triangle's normal is pointing is determined by the order in which the vertices have been specified. Triangles specified with vertex order consistent with the right hand rule or counter-clockwise point in the direction of the thumb. This ordering is arbitrary for single triangles as can be seen in Figure 6.2, $v1$->$v3$->$v2$ is counterclockwise from the view of $\overrightarrow{E_1}$, $v1$->$v2$->$v3$ although clockwise from $\overrightarrow{E_1}$ is counterclockwise from the view of $\overrightarrow{E_2}$.

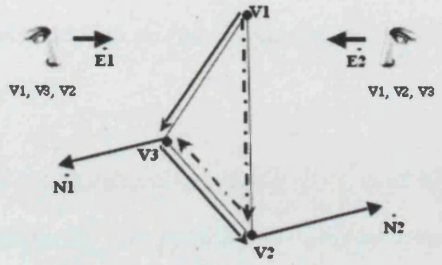


Figure 6.2: Counter-clockwise specification exists on both sides of a triangle.

Consequently for an *open surface*[1] this can make the surface either totally visible or not rendered at all (Figure 6.3).

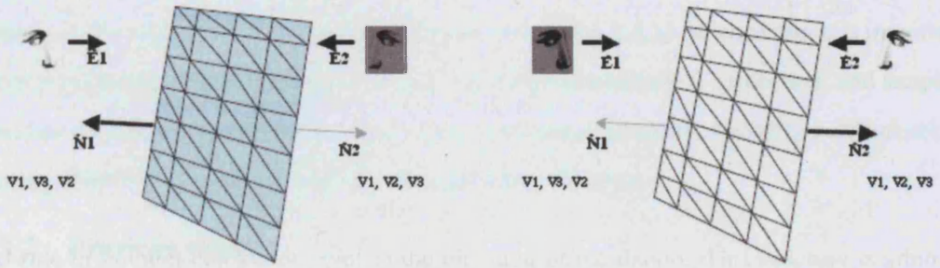


Figure 6.3: Surface *culling*.

With a *closed surface*[2] the decision is no longer arbitrary, because there is an orientation where all the triangles are specified to point towards the outside and thus the exterior is rendered when back-face culling. Therefore it becomes important to reliably determine what direction is

[1] A collection of connected triangles surrounded by border edges.

[2] A collection of connected triangles that is not surrounded by border edges, possibly containing some holes with border edges.

inwards, and what is *outwards* (Figure 6.4 shows normals incorrectly pointing *inwards*, hence making part of the interior visible while other parts nearer to the viewer have been culled).

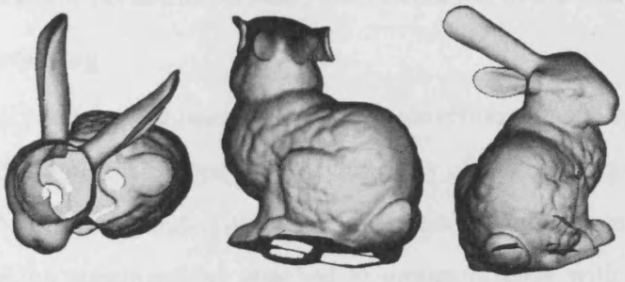


Figure 6.4: An *inwards* oriented object. The original model was made available by the Stanford Computer Graphics Laboratory.

6.1.1.1 The impact of orientation inconsistency on level of detail algorithms

Only LoD methods that do not use the orientation of triangle normals in the cost function they are trying to minimise remain unaffected by orientation inconsistency. For example, an LoD method that just uses the length of edges as a cost function is not affected. Similarly the quadric error approach [GS97] uses the unsigned distance to a plane/triangle and thus numerical results obtained using it are not affected.

However the method of simplification envelopes [CVM+96] uses the triangle normals to build the outer and inner envelope around a surface and is therefore severely compromised (a review of this method was presented in Chapter 2, Section 2.4.1). In any case, it is important to have consistently oriented surfaces not just for robust visualisation, inspection, and simplification but for other algorithms that require this consistency property, visibility culling/back face culling [Gil07] and *radiosity*[3] calculation being fine examples.

6.1.2 Previous work

There is not much published work on fixing inconsistent normals. There are commercial tools available but unfortunately no details on their operation or quality when dealing with real data such as that of laser scans. However it is not infeasible that such methods take an approach similar to that of the freely available tool ivnorm which does have algorithm details published

[3]This is "a technique that calculates the lighting in a complex diffuse-lighting environment, based on the scene's geometry. Because the radiosity calculations do not include the eye point of the viewer, the geometry and lighting in the environment do not need to be recalculated if the eye point changes. This enables the production of many scenes that are part of the same environment (the rooms in a building, for instance), and a "walk through" the environment in real time" [Sun05].

in [Bel95]. Instead of a solution in which the orientation of individual triangles are determined one at a time, triangles that share vertices with adjacent triangles are first *grouped* into a larger surface and second, *a test* is performed to verify the orientation of the whole group.

6.1.2.1 Normal grouping

Some solutions [R307] require the user to choose the directions manually. This can be quite tedious for the user if the object is composed of hundreds of patches, dispersed across the entire model. Figure 6.5 shows groups/patches of connected triangles correctly rendered with counter-clockwise vertex specifications attached to groups/patches with incorrect clockwise vertex specified triangles. The model corrected by our algorithm which will be presented later in this chapter is reported in Figure 6.29, rightmost column).

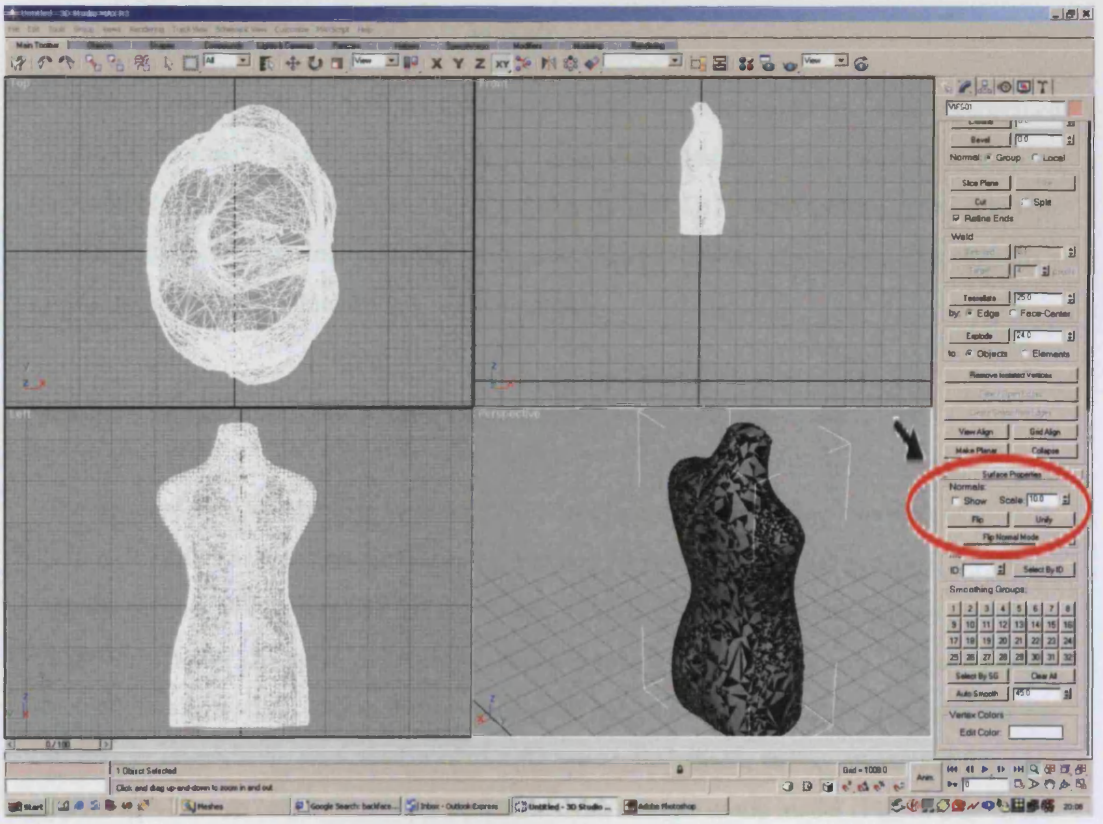


Figure 6.5: 3DStudioMax: manual normal fixing.

ivnorm will attempt automatically to force a particular vertex ordering accross all connected triangles in the model. After this operation, a surface will typically consist of just one large group/patch. However the presence of non-manifold edges[4] (Figure 6.11) creates a prob-

[4] A non-manifold edge is an edge that is shared by more than two triangles and occur where two or more distinct surfaces make contact by sharing the same particular vertex pair. At these locations any of the surfaces involved are said to be non-differentiable, as it is not clear what the continuation of the given surface is.

lem when performing surface connectivity queries such as determining the adjacent connected triangle of another triangle at an edge. This can lead to two problems.

Firstly, some triangles can become inaccessible therefore creating fragmented smaller groups which in turn increase the computational demands of determining a greater number of correct orientations for each of these groups. Secondly, and more importantly, if these triangles are accessible they can inconsistently propagate the vertex-order criterion or direction within the same surface group, effectively turning the surface in on itself.

In surface scans these degeneracies are often the line of contact of several small noisy surfaces. At these problematic locations, even if all triangles are accessible it becomes difficult without more information about the object to establish which triangles belong to the current surface being grouped as the choice is arbitrary.

Similarly *plyorient [ply05]* is a freely available tool that attempts to orient objects specified in the PLY format[5]. Unfortunately if there are non-manifold edges or non-manifold vertices[6] in a model *plyorient* reports duplicate edges and stops (Figure 6.17, *left* illustrates a particular type of non-manifold vertex that is formed when more than two border edges join). This suggests a problem in how the tool represents edges in memory.

6.1.2.2 Establishing the orientation of a group with a ray test

The standard way to determine whether a closed surface/patch is pointing *outwards* or *inwards* is to pick a triangle from that surface and use its normal to fire a ray in that direction. By counting the number of intersections this ray makes with triangles in the model one can attempt to establish whether the given normal was already correctly pointing *outwards*: an odd number of triangle hits would indicate the ray intersected only the starting triangle that provided the normal; or *inwards*: an even number of triangle hits would indicate the ray intersected the starting triangle that provided the normal and some other triangle in the model (Figure 6.6). If the normal was found to be pointing *inwards*, the vertex specifications of each triangle belonging to that surface need to be reversed in order to make the normals point *outwards* (recall Figure 6.3).

[5]PLY format is a simple object/polygon description that was developed at Stanford University.

[6]A non-manifold vertex exists when two or more distinct surfaces make contact by sharing a single common vertex.

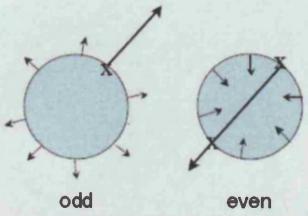


Figure 6.6: Odd/even counting

Techniques such as *ivnorm* that simply use a triangle normal to intersect the rest of the object and rely on counting surface intersections of rays to determine inside/outside directions face a difficult problem with non-manifold and self-intersecting surfaces. Quite often clusters of degenerate surfaces float inside an object where for example, the surface did not have enough sample data. We compare our results with *ivnorm* [Bel95] in Section 6.1.5.

6.1.2.3 Identified pitfalls

The following can have an adverse effect on the counting of ray intersections test described in the previous section. The last three items can determine the robustness of any strategy that attempts to establish orientation consistency of models in general.

- The likelihood that a ray will hit an edge and count both triangles sharing that edge.
- Multiple counting of triangles when a ray hits a surface at a tangential angle.
- The starting point of a ray.
- Small floating surfaces.
- Holes in the surface.
- Non-manifold edges.
- Non-manifold vertices.
- How edges are represented in the system.

The likelihood that a ray will hit an edge and count both triangles sharing that edge

As model complexity increases the chance that a ray is going to hit an edge will also increase as can be seen in Figure 6.7.

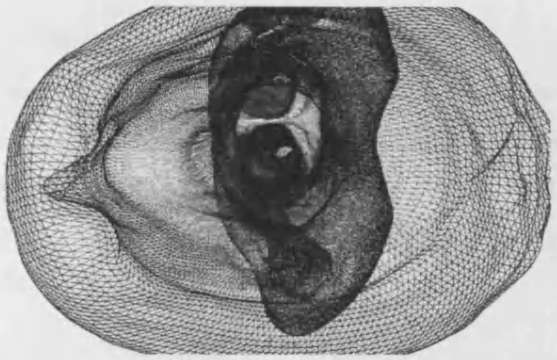


Figure 6.7: The likelihood that a ray will hit an edge increases as model complexity increases and multiple parts of a dense scanned mesh intertwine the ray path.

The consequence of hitting an edge is typically that the the two triangles sharing the edge will fall within the tolerance of a positive decision that there has been an intersection with the ray, hence counting as two triangle hits instead of one hit for the surface. If the normal were pointing inwards, this situation would incorrectly have an odd number of triangle intersections as the starting triangle is always counted. The test will report a correctly oriented surface when it should instead be inverted. This double counting is illustrated in Figure 6.10, left.

Multiple counting of triangles when a ray hits a surface at a tangential angle

Any and all triangles in a group hit tangentially by a ray will adversely contribute to the counting test. A classical problem in *raytracing* is illustrated in Figure 6.8. In this situation five triangles are hit, but this should only count as one hit.

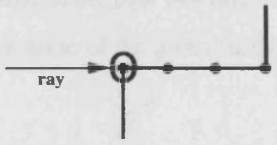


Figure 6.8: Ray-multiple face hit

Raytracing according to [Sun05] is: "An advanced method of determining light interaction, such as reflection and refraction, in a graphical lighting environment. Rays are traced from the light source to the eye point, or from the eye point to the light source, to determine what the eye sees through each pixel on the display. Ray tracing mainly yields realistic results, but it is computationally intensive".

The starting point of a ray

We mentioned that the starting triangle from where the ray is fired is always counted as one hit, however if the ray is generated on an edge, again two triangles are reported instead of one for the surface. Therefore a tempting strategy is to simply fire the ray from the centre of a given triangle, but with symmetrical objects this can have significant drawbacks, as illustrated in Figure 6.9.

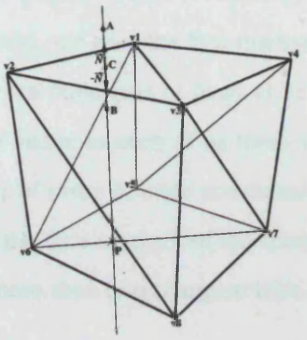


Figure 6.9: Ray starting point/ray-edge hit

Holes in the surface

If the surface has holes, then the rays used in the ray test could potentially go through them, leaving surfaces unaccounted for and incorrectly affecting the number of intersections in the test.

Small floating surfaces

If degenerate small surfaces are found in the path of a ray, the number of hits will again become unreliable for determining the orientation of the given surface (Figure 6.10).

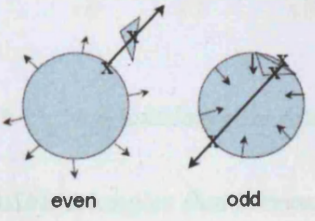


Figure 6.10: Double counting resulting of intersection of small floating surface.

Non-manifold edges

As mentioned previously, non-manifold edges can create two problems:

- *Small extra groups of inaccessible triangles that increase computation demands*
- *The folding of a surface on itself, that lends it to be only partially visible*

To better understand how these problems can occur, let us first examine how a non-manifold edge can be detected. Let us use the connectivity/marking data structure adopted by [Gar99], which requires significantly less memory than a winged edge data structure[Bau74] (edges of a model are implicit rather than explicitly stored in memory). Figure 6.11 illustrates the procedure. For every edge in the model, we zero the face markers of every triangle connected to each of the edge's vertices. One can build lists of faces at vertex level by taking one triangle at a time, and adding the triangle index to each of its three vertices own face lists. Next we increment by one the face markers of every triangle associated with the first vertex of the edge. Finally we increment by one all the face markers of the triangles associated with the second vertex of the edge. If there are more than two triangles with a face mark value of 2, then we have a non-manifold edge.

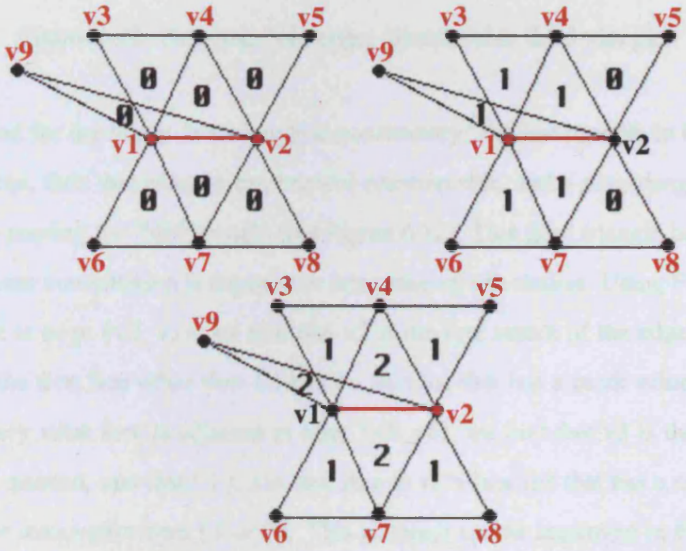


Figure 6.11: Non-manifold edge detection

Small extra groups of inaccessible triangles that increase computation demans

We can use the marking procedure described above to find the triangles adjacent to each other at an edge. Edges are typically shared by only two triangles. However if there are more than two faces sharing the edge, then the triangle deemed to be adjacent to another will depend on the order the triangles appear on the vertex face list of the edge. The triangle retrieved in the query will be the first face in the vertex's face list that has a mark value of two and which is not

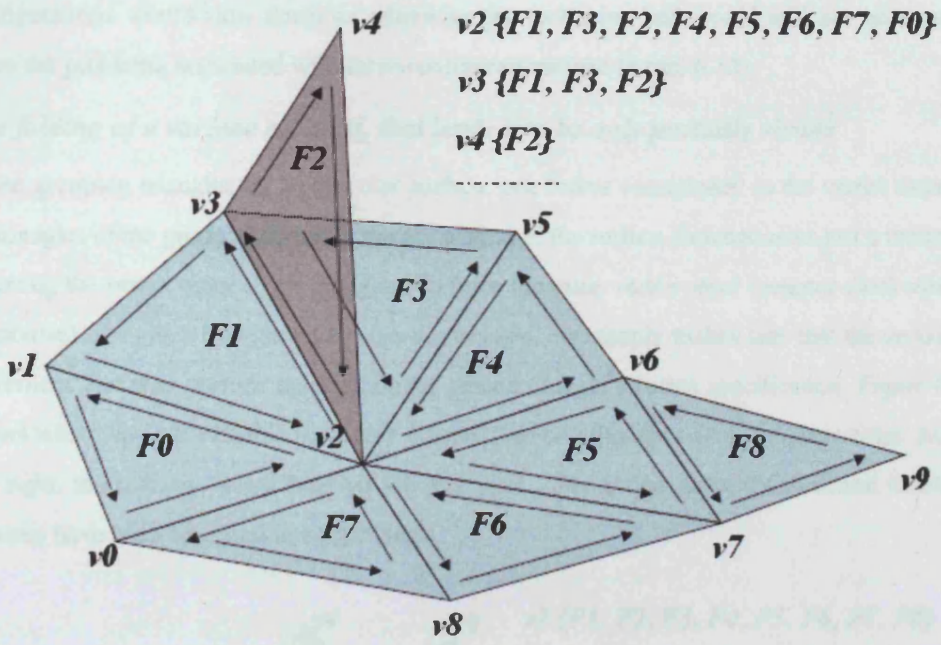


Figure 6.12: Non-manifold edge; unretrievable third triangle.

the face provided for the query. If a triangle is consistently[7] the last triangle in both facelists of an edge's vertices, then that triangle can become unretrievable, and a *ping-pong* like relation is established, by-passing the third triangle (see Figure 6.12). This third triangle becomes a group by itself, and more computation is required to determine its orientation. Using F1 to query what face is adjacent at edge {v2, v3}, we find that v2 is the first vertex of the edge being queried, and that F3 is the first face other than F1 in v2's face list that has a mark value of 2. Then by using F3 to query what face is adjacent at edge {v3, v2}, we find that v3 is the first vertex of the edge being queried, and that F1 is the first face in v3's face list that has a mark value of 2. Hence F2 is not retrievable from F1 or F3. This situation can be improved in the case of three triangles sharing an edge by changing the way the vertex face lists are created. One could go through each triangle three times, adding one vertex at each pass. This will change the order in which faces with a mark value of two appear on the vertex face lists, enabling some triangles to be accessible. However, with configurations where four or more triangles share the same edge, this strategy makes it impossible to access all triangles. This connectivity query strategy has proven to be robust in non-degenerate models, and changing it to cope with problematic

[7] recall that the vertice's face lists are built by adding one triangle at a time at a time to each of its vertices. This operation has the property that the order triangles are added at adjacent vertices is the same, and its therefore consistent.

configurations would slow down an otherwise fast technique and would still not necessarily solve the problems associated with these configurations (see Figure 6.13).

The folding of a surface on itself, that lends it to be only partially visible

When grouping triangles on a particular surface, one forces consistency in the vertex order in all triangles of the group. Correcting the orientation of the surface then becomes just a matter of inverting the vertex order of all triangles. To force the same vertex order (counter-clockwise or clockwise) of a given triangle on an adjacent triangle, one simply makes sure that the sequence of vertices that they share is reversed on the second triangle's vertex specification. Figure 6.13 shows what happens with this procedure if more than two triangles share the same edge. At the top right, the relevant vertex facelists are displayed. The arrows show the direction in which vertices have been specified in each triangle.

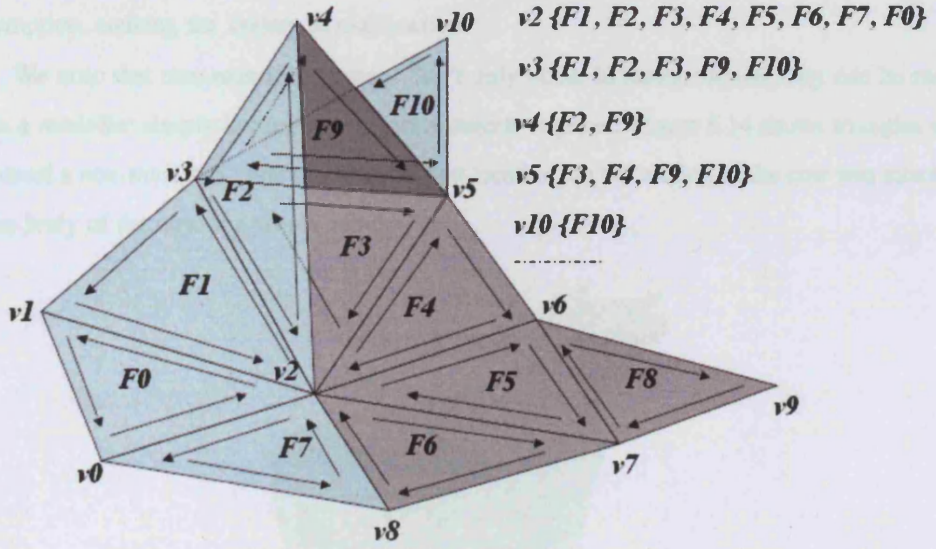


Figure 6.13: Non-manifold ege and inconsistent vertex order propagation.

Starting with the counter-clockwise triangle F1 {v1,v2,v3}, one performs a connectivity query at edge {v2,v3}, looking up the first vertex of this edge (in this case v2). We find that the first face other than F1, marked with a value of two in v2's facelist is F2{v2,v3,v4}. We make sure that the vertex sequence at that edge is reversed, making F2{v3,v2,v4}. We proceed with F2 to query what is triangle adjacent at the edge {v4,v3}; looking up v4, we find that it is F9, this forces F9 to be counterclockwise{v3,v4,v5}. We then query at edge {v5,v3} to find what is the next connected triangle, and we find by looking at v5 that F3 is the first face with a mark of two. It is here that the inversion of the surface takes place, by forcing F3 edge to be

{v3,v5}, the reverse of F9's edge {v5,v3}, we have effectively reversed the order of the vertices from counter clockwise to clockwise. A depth first strategy will continue propagating this order from F3 to F4, F5, F6, F8, etc. A breadth-first strategy will suffer less, but will give an equally unsatisfactory result. The triangles shown in the figure are part of the same group, F7 and F6 are manifold at v2 and v8 but have different orientations.

We note that v4 is typically a spike/noise that gets attached to a surface during the surface reconstruction process.

Non-manifold vertices

If a system makes only surface connectivity queries at edges of triangles to span a surface, then some triangles could be irretrievable with this approach at non-manifold vertex locations. Also, if a system assumes that a vertex index appears only once for a set of triangles surrounding it, then the presence of a second, different set of triangles around the vertex could invalidate that assumption, causing the system to malfunction.

We note that non-manifold vertices don't only occur in surface scans, they can be result from a modeller simply linking two separate objects together. Figure 6.14 shows triangles that surround a non-manifold vertex in blue. At that location the tail section of the cow was attached to the body of the cow at a single vertex.

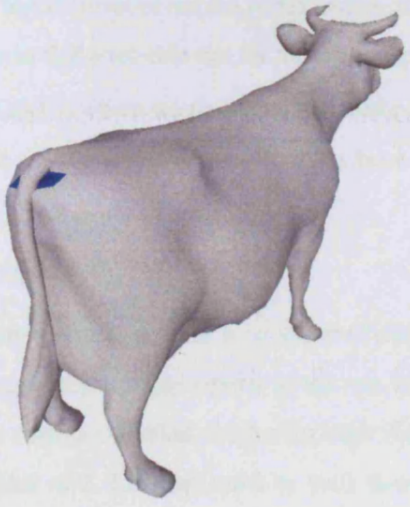


Figure 6.14: Non-manifold vertex where the tail object is attached to the main body surface.

How edges are represented in the system

The connectivity data structure discussed above does not explicitly store edges in memory. However, such a system might choose to store them if other operations are going to be per-

formed on the data, such as geometry reduction (Chapter 4).

If two adjacently connected triangles have a *consistent vertex ordering*, then the edge that they share will have one vertex order at one triangle and the reverse vertex order at the other. For example, in Figure 6.13 triangle F0{v0,v2,v1} will have the edge {v2,v1} and triangle F1{v1,v2,v3} will have the edge {v1,v2}. In the case of adjacently connected triangles with different vertex ordering specification, for instance F7{v0,v8,v2} and F6{v8,v2,v7}, one or more edges will appear twice with the same vertex order, in the example it will be edge {v8,v2}. If we are attempting to establish orientation consistency in a model, it would appear plausible that the same edge with the same vertex specification will occur twice or more in the case of non-manifold configurations. If the system requires distinct edges per triangle or per vertex, then this can somewhat limit the models it is able to fix.

6.1.3 Determining the orientation of 3D models

Our algorithm has four distinct phases. The first one triangulates holes in a model, ensuring that rays can't go through them. The second addresses the problem of non-manifold edges by either marking them out of consideration, or by repairing them. The third phase creates normal groups and forces a consistent vertex ordering across the group, problematic marked triangles are not grouped. Finally the fourth phase recomputes the normals of the new vertex orders and uses the normals in our ray test to invert or not the vertex orders of the groups. Rays of opposite orientation are used in pairs to find a reliable test for the normal group direction.

This work builds on [OS02], where we presented our method for reliably determining the orientation of laser scanned surfaces, but the algorithm has been improved to handle multiple objects and large models.

6.1.3.1 Hole triangulation

The first step of the hole triangulation process is to retrieve border edges, these border edges can be found with a marking/query strategy similar to the one illustrated in Figure 6.11, with the difference being that an edge is classified as a border edge if there is only one triangle that shares the edge with the value of 2. Lists are made to track these detected border edges, and they are sorted according to the smallest index value of the vertex pair. This allows one to easily follow a connected edge sequence in the list. When the sequence is broken, e.g. an edge shares no values with the previous edge in the list, this indicates the start of a different hole in the model. Connecting the border vertices to the centroid (average vertex position of the vertices in an edge sequence) does not ensure non-self intersection of the resulting surface. However, rays that would otherwise go through holes, and have multiple intersections with a self-intersecting

surface are successfully deemed to be unreliable in our ray test (see Section 6.1.3.4), and new rays will be created. The Stanford bunny is a model that exhibits large holes at its base. When determining the orientation for this model, if a ray were to be fired inwards into the model, one would expect an even number of intersections. However, if the ray were to pass through the hole it would detect an odd number of intersections, potentially inverting a surface that was already correctly oriented. An example of such a ray would be one starting from the ear of the bunny correctly pointing outwards and towards the main body of the scan, the surface of the body would be intersected, but the ray could exit the body through the hole. Consequently, an even number of intersections would be reported and would ultimately result in the normals of the model being inverted. Figure 6.15 shows the holes in the Stanford bunny triangulated with the simple centroid strategy. We recall that the primary purpose of triangulating holes is to detect and count any ray that might go through a hole. Since triangulated holes are typically deleted by default after the object is oriented, we did not incorporate mesh quality constraints that are typically required for avoiding rendering artefacts when rendering the model.

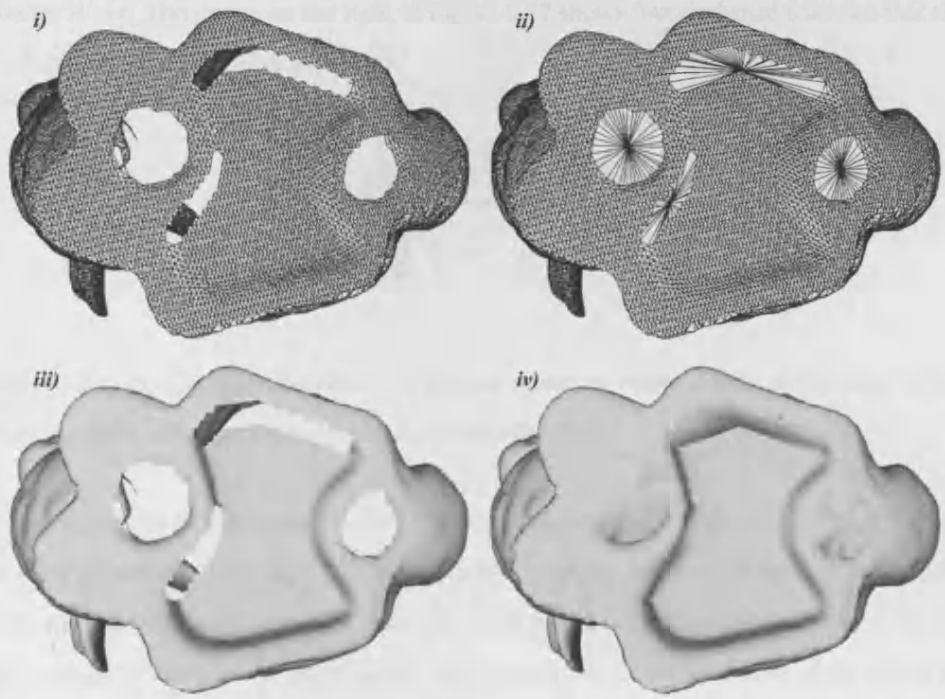


Figure 6.15: Triangulating holes.

It would be desirable to correct initial non-manifold configurations by deleting them and triangulating the resulting holes. One way to try to achieve this would be to only triangulate a sequence of connected border edges whose border triangles were classified into the same

normal group. This situation is illustrated in Figure 6.16, where new triangles are created with the vertices of the border edges and the centroid P.

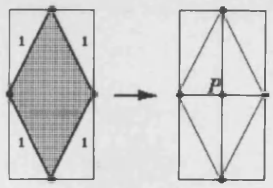


Figure 6.16: *Left:* border vertices with border-edge valence of 2, *right:* hole triangulation.

Unfortunately, hole triangulation does not ensure non-manifold edge creation, hence this operation needs to occur before the non-manifold edge repair or detection phase. Figure 6.17 shows a connected edge border sequence whose border triangles were tagged to the same normal group *1*. The vertex A has a valence of 4 border edges connected to it, instead of 2 as for border vertices in Figure 6.16. Although the border edge sequence is valid, the border vertex sequence is not. The image on the right of Figure 6.17 shows four darkened triangles that share the resulting non-manifold edge *P-A*.

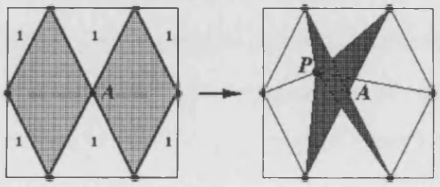


Figure 6.17: *Left:* Complex Boundary vertex/non-manifold vertex A with border-edge valence of 4, *right:* hole triangulation with non manifold edge PA.

We have tried to triangulate sequences that stopped at border vertices with valences higher than 2 (non-manifold vertices). But unfortunately in all the scanned models, all the resulting border vertices have a valence of 4. An object that similarly exhibits this property is the Sierpinski triangle (Figure 6.18), where all but the three corners on the silhouette of the object have a border edge valence of 4. It is not clear what benefits other hole triangulation schemes such as [Hel01] and [SZL92] can offer in this situation. Border vertices that have a border-edge valence higher than 2 are likely to create non-manifold configurations. For completeness we would like to add the vertex classification: *Complex Boundary* to Schroeder's five: *Simple*, *Complex*, *Boundary*, *Interior Edge*, *Corner*. We note that a *Complex Boundary* is a type of non-manifold vertex created through the decimation of an edge, and although no non-manifold

edges are connected, they are complex. There is little point in deleting non-manifold edges if they are going to be created again in the hole triangulation process. In the end we chose not to delete them, and instead mark or repair them after the hole triangulation is complete.

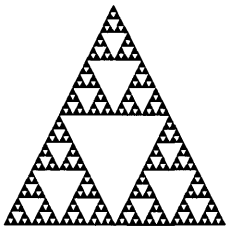


Figure 6.18: Sierpinski triangle, border edge valence > 2

We note that in the case of open surfaces, such as a terrain (Figure 6.21), there would be no benefit in triangulating the hole made by the borders of such object. Indeed, an open surface such as a mobius strip (Figure 6.28) would have problems if the hole formed by its borders were to be triangulated, as the surface changes orientation on its border. Hence our algorithm has two main modes, it defaults to closed objects that may have holes, or alternatively it lets the user specify whether the object being treated is an open surface, and no holes are triangulated. The user also has the option of saving the triangles used to cover holes.

6.1.3.2 Repairing non-manifold configurations

After triangulating any existing hole, we proceed to repair or mark non-manifold edges. After the object has been oriented, non-manifold vertices and edges need to be removed or repaired for LoD processing. We cover non-manifold vertex repair here, even though these vertices do not affect our re-orientation algorithm.

Repairing non-manifold vertices

A non-manifold vertex essentially has more than one set or group of triangles connected to it. Figure *6.19-i)* shows a non-manifold vertex A, vertex B is a manifold vertex. If we attempt to repair vertex A by creating new vertices for each triangle connected to A, we successfully repair A but unfortunately cause vertex B to become non-manifold Figure *6.19-ii)*. Algorithm 7 & 8 outlines the solution illustrated in Figure *6.19-iii)*.

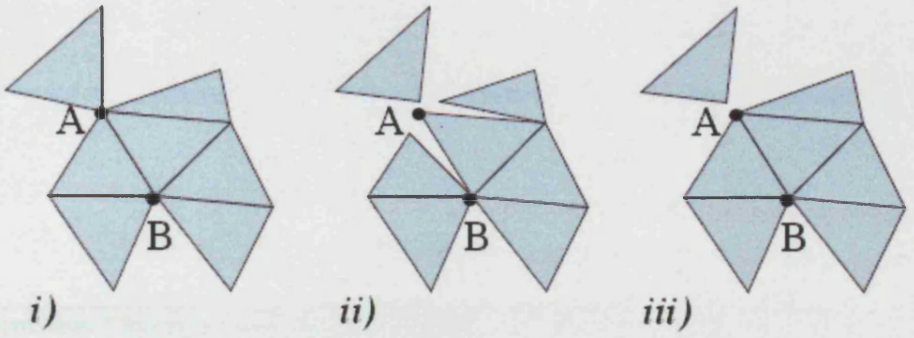


Figure 6.19: Repairing non-manifold vertices

All connected faces are recursively assigned the same group id. When the recursion stops and there is still a face left unmarked, then we are in the presence of a non-manifold vertex and proceed to recursively mark further unmarked faces with a different group id. Now that the faces have been marked with their relevant group id, we can create one new vertex for each face group. These new vertices can have exactly the same coordinates of the original non-manifold vertex, or be offset by some scalar. Some care is needed in the choice of offset, as a fixed and relatively small floating point offset will not affect a vertex expressed in a larger scale representation. We found that using a value equal to 1/20th of the average edge length in the model overcomes such floating point problems. Another alternative is to simply have no offset, the non-manifold condition exists through connectivity and indexing, and if the indexing is changed then the position of the new vertices becomes somewhat irrelevant. However if vertices are compared for uniqueness in 3D space and duplicates cleaned, then choosing an offset would be a better choice as the non-manifold condition will re-emerge if a zero offset was chosen.

Repairing non-manifold edges

Non-manifold edges can be detected with the marking scheme illustrated in Figure 6.11. Figure 6.20-*i)* shows a non-manifold edge C-D, with associated non-manifold triangles shaded in grey. As mentioned in Section 6.1.3.1, simply deleting non-manifold faces associated with non-manifold edges can create holes that are difficult to re-triangulate in a non-manifold free way, Figure 6.20-*ii)* shows the resulting hole from deleting non-manifold faces, and how vertex D became a non-manifold vertex.

Algorithm 7 Repairing non-manifold vertices.

```
function repair_allnonmanifold_vertices(offset)
    for(vid=0 vid<number_of_vertices vid++)
        p=atVertexArray(vid)
        p->set_fv(1) // number of times the vertex is used
    for(vid=0 vid<number_of_vertices vid++)
        p=atVertexArray(vid)
        fgroups=0
        for(face ∈ p->flist)
            face->set_groupid(0)
        for(face ∈ p->flist)
            if(face->get_groupid()==0)
              fgroups++
              add_to_facegroup_aroundvertex(face, vid, fgroups)
        if(fgroups>1) // the vertex is non-manifold/repair it
            x=p->X(); y=p->Y(); z=p->Z();
            for(ii=2 ii<=fgroups ii++)
              if(offset!=-1)
                 shiftno=p->get_fv()
                 p->set_fv(shiftno+1)
                 x=x+shiftno*offset;
                 y=y+shiftno*offset;
                 z=z+shiftno*offset;
              newvid=insertPoint3D(x,y,z)
              newp=atVertexArray(newvid)
              for(face ∈ p->flist)
                 if(face->get_groupid()==ii)
                  newp->addfaceref(face)
                  face->replacevertexref(vid,newvid)
                  templist->addface(face)
            for(face ∈ templist)
                 p->deletefaceref(face)
```

Algorithm 8 Repairing non-manifold vertices (cont.).

function add_to_facegroup_aroundvertex(face, vid, gid)

```
if(face->get_groupid()==0)
    face->set_groupid(gid)
    indA=face->firstvertex()
    indB=face->secondvertex()
    indC=face->thirdvertex()
    if(indA==vid)
        find_facessharingedge(indA,indC,flist)
    else if(indB==vid)
        find_facessharingedge(indB,indA,flist)
    else
        find_facessharingedge(indC,indB,flist)
    if(indA==vid)
        find_facessharingedge(indA,indB,flist)
    else if(indB==vid)
        find_facessharingedge(indB,indC,flist)
    else
        find_facessharingedge(indC,indA,flist)
    for(face ∈ flist)
        if(face->get_groupid()==0)
            add_to_facegroup_aroundvertex(face,vid,gid)
```

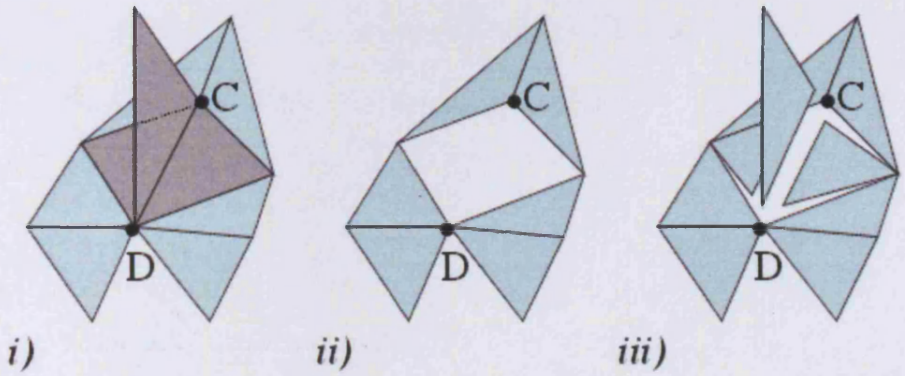


Figure 6.20: Repairing non-manifold edges.

One way to prevent non-manifold edge creation is described in Algorithm 9. We first use the marker of each face to mark which edges of the triangle are non-manifold. We then proceed by adding new vertices for each face accordingly (Figure 6.20-*iii)*). This strategy will still make the vertex D non-manifold; we note however that our non-manifold vertex repair algorithm does not introduce degeneracies. Hence, to make a model free from non-manifold edges and vertices, one needs to call the algorithm for non-manifold edge repair first, and then

call the non-manifold vertex repair algorithm to repair any original non-manifold vertices or those newly created ones from the edge repair process. Having an offset of zero will ensure that new holes are not created, thus removing the possibility that rays will pass through the model in our ray test.

Algorithm 9 Repairing non-manifold edge regions.

```
function repair_allnonmanifold_edge_regions(offset)
    for(fid=0 fid<number_of_faces fid++)
        face=atFaceArray(fid) face->set_groupid(0)
        // face groupid key:
        // 1 means 1stedge, 11 means 2ndedge and 1st
        // 10 means 2ndedge, 100 means 3rdedge
        // 101 means 3rdedge and 1st
        // 111 means all edges
    for(vid=0 vid<number_of_vertices vid++)
        // number of times the vertex is used
        p=atVertexArray(vid);p->set_fv(1)
    for(fid=0 fid<number_of_faces fid++)
        face=atFaceArray(fid)
        v1=face->firstv() v2=face->secondv() v3=face->thirdv()
        res1=0 res2=0 res3=0
        res1=isnonmanifoldedge(v1, v2)
        if(res1==1)
            if(face->get_groupid()==0)        face->set_groupid(1)
            else if(face->get_groupid()==10)  face->set_groupid(11)
            else if(face->get_groupid()==100) face->set_groupid(101)
            else if(face->get_groupid()==110) face->set_groupid(111)
        res2=isnonmanifoldedge(v2, v3)
        if(res2==1)
            if(face->get_groupid()==0)        face->set_groupid(10)
            else if(face->get_groupid()==10)  face->set_groupid(11)
            else if(face->get_groupid()==100) face->set_groupid(110)
            else if(face->get_groupid()==110) face->set_groupid(111)
        res3=isnonmanifoldedge(v3, v1)
        if(res3==1)
            if(face->get_groupid()==0)        face->set_groupid(100)
            else if(face->get_groupid()==10)  face->set_groupid(110)
            else if(face->get_groupid()==100) face->set_groupid(101)
            else if(face->get_groupid()==110) face->set_groupid(111)
    for(fid=0 fid<number_of_faces fid++)
        face=atFaceArray(fid)
        v1=face->firstv() v2=face->secondv() v3=face->thirdv()
        gid=face->get_groupid()
        if(gid==1)         vlist->add(v1) vlist->add(v2)
        else if(gid==10)   vlist->add(v2) vlist->add(v3)
        else if(gid==100)  vlist->add(v3) vlist->add(v1)
        else if(gid==11)   vlist->add(v1) vlist->add(v2) vlist->add(v3);
        else if(gid==101)  vlist->add(v1) vlist->add(v2) vlist->add(v3);
        else if(gid==110)  vlist->add(v2) vlist->add(v3) vlist->add(v1);
        else if(gid==111)  vlist->add(v1) vlist->add(v2) vlist->add(v3);
        if(gid!=0)
            for(vid ∈ vlist)
              p=atVertexArray(vid)
              x=p->X(); y=p->Y(); z=p->Z();
              if(offset!=-1)
                 shiftno=p->get_fv()
                 p->set_fv(shiftno+1)
                 x=x+shiftno*offset;
                 y=y+shiftno*offset;
                 z=z+shiftno*offset;
              newvid=insertPoint3D(x,y,z)
              newp=atVertexArray(newvid)
              newp->addfaceref(fid)
              face->replacevertexref(vid,newvid)
              p->deletefaceref(fid)
```

6.1.3.3 Normal group creation

The previous phase repaired non-manifold edges, alternatively one can mark the detected triangles out of consideration for normal fixing.

Once the triangles associated with non-manifold edges are repaired or marked out, we pick the first triangle of the object that has not been marked non-manifold and does not yet belong to a group. We mark it with the current number of the group, retrieve the three adjacent triangles and force the vertex order on them to be consistent with the picked triangle, recursively applying the same procedure to the adjacent triangles that have not yet been marked. When there are no more connected triangles the recursion will stop and return to the main loop, where the triangles that have been marked are skipped until an unmarked one starts a new group (Algorithm 10)

Algorithm 10 Pseudo-code of normal grouping, *recursive version.*

```
function fixallnormals(mode)
    fgroup=1
    if(mode==1)
        // mark all non-manifold face group with -1
        mark_allnonmanifold_edge_regions()
    else
        repair_allnonmanifold_edge_regions()
    for(i=0 i<number_of_faces i++)
        face=atFaceArray(i)
        if(face->get_groupid()==0)
            // first ever vertex order specification,
            // and not non-manifold part
            fixvertexnormals(face,fgroup)
            fgroup++

function fixvertexnormals(face, fgroup)
    face-> wt = fgroup
    v1id = face->firstvertex()  v2id = face->secondvertex()  v3id = face-
    >thirdvertex()
    nf1=getadjacentfaceatedge(v1id, v2id, face)
    forcevertexorder(face, nf1)
    nf2=getadjacentfaceatedge(v2id, v3id, face)
    forcevertexorder(face, nf2)
    nf3=getadjacentfaceatedge(v3id, v1id, face)
    forcevertexorder(face, nf3)
    if(nf1->get_groupid()==0)
        fixvertexnormals(nf1,fgroup)
    if(nf2->get_groupid()==0)
        fixvertexnormals(nf2,fgroup)
    if(nf3->get_groupid()==0)
        fixvertexnormals(nf3,fgroup)
```

We note that by ensuring that all triangles are visited in the main for loop in *fixnormals* (Algorithm 10), we avoid the problems that non-manifold vertices can create (some triangles

may become unreachable through connectivity queries if we were only accessing triangles connected to edges). This strategy has proved to be a strong feature in the design of our algorithm, and the concept is the basis for our non-manifold vertex repair Algorithm 7.

Although this recursive grouping works fine with models of a few thousand triangles, its depth first approach will exhaust a computer's stack memory quickly with function arguments and local variables. To handle large models such as the Turbine Blade model (1.7 million triangles, Figure 6.32, right), an iterative solution was developed and is explained in Section 6.1.4, Algorithm 11.

6.1.3.4 Reliable ray tests

The previous phase created groups of connected triangles and forced a consistent vertex ordering in all triangles of each group. At this point, it is important to recalculate the normals of all triangles since they may have been flipped when forcing vertex order. Our ray test will use one of these normals to either invert or not all normals in the group. On average, the grouping phase successfully groups 98% of the model into one surface group (Table 6.4, 2^{nd} and 4^{th}column). Given that one test can potentially determine the orientation of the whole group, this allows for some freedom to choose the most reliable test rays for all our tests.

For simplicity a ray is defined with two points: a starting point (Cx, Cy, Cz) and a second point (Bx, By, Bz)which defines the direction of the ray following a particular triangle orientation $\vec{N}$ or -$\vec{N}$ (Figure 6.9). A point (Px, Py, Pz) on the ray can be found with the following equations:

$$\begin{aligned} Px &= Cx + \alpha * (Bx - Cx) \\ Py &= Cy + \alpha * (By - Cy) \\ Pz &= Cz + \alpha * (Bz - Cz) \\ & \quad where\ 0 \leq \alpha < +\infty \end{aligned} \tag{6.1}$$

In our case we are interested in finding a point on the ray that intersects a plane:

$$a * (Px) + b * (Py) + c * (Pz) + d = 0 \tag{6.2}$$

substituting Equation 6.1 in Equation 6.2

$$a * (Cx + \alpha * (Bx - Cx)) + b * (Cy + \alpha * (By - Cy)) + c * (Cz + \alpha(Bz - Cz)) + d = 0$$

and solving for α:

$$\alpha = \left(\frac{-a * (Cx) - b * (Cy) - c * (Cz) - d}{a * (Bx - Cx) + b * (By - Cy) + c * (Bz - Cz)} \right) \tag{6.3}$$

Care needs to be taken with the denominator of Equation 6.3, as the ray might be parallel to the plane and not intersect it, yielding a zero dot product between the plane normal and the ray's orientation.

The outline of our test strategy is as follows:

1. For each normal group, choose the first triangle in the group to fire a ray from. If the triangle is thin or degenerate pick the next one in the same group, if the group only has thin triangles, do not orient this group.

2. For the picked triangle create 3 random barycentric coordinates for the starting point C of the ray. Make sure the random point is not on one of the edges of the triangle as this would count as two intersections. Continue to create random barycentric coordinates if they fall on an edge. Step 1 ensured thin triangles were not considered, as they could make it impossible to fire a ray that was not on one of the triangle's edges.

3. Use the triangle normal $\vec{N}$ to calculate the second point A, that determines the direction of the ray.

4. Intersect the ray with all the triangles of the model. If you have a spatial data structure, query the data structure. If the ray hits an edge, go back to 2 with the same triangle. If not, record the number of hits (hitsA). If hitsA is one, proceed with the next group. Go to step 1.

5. Use the triangle normal -$\vec{N}$ to calculate a ray with the opposite direction of the one created in 3.

6. The same as 4, compute two new rays for the same triangle if it hits an edge. If not, record the number of hits separately (hitsB).

7. Check to see if either hitsA or hitsB has the value of one. If hitsA has a value of one, it means that the group was oriented correctly, and we proceed with a triangle of the next group. If hitsB has a value of one but hitsA has not, then we reverse the vertex order for all the triangles in the group, and proceed to the triangles of the next group.

8. If neither hitsA or hitsB has a value of one, we go back to step 2, with another triangle of the same group. Hopefully this new triangle will be positioned in a more reliable

location, away from self-intersecting surfaces. In principle, with surface models, it should be possible to find a triangle in the group where one of the rays hits only one triangle, the one it started from. We also keep a count of how many triangles we have tried, and if we have tried all the triangles in the group, we reason with the smallest value of hitsA and hitsB. If hitsA has the smallest value, and it is odd, then we proceed with the triangle of the next group. If it was even, we invert the vertex order of all the triangles in the group, as we do if hitsB had the smallest value and it is odd. Finally, if hitsB has the smallest value and it is even, we do not invert the vertex order of the group and proceed to step 1.

Care needs to be taken to avoid double counting of triangle intersections when a ray hits an edge. Systematically shooting the rays from the centre of a triangle is a bad strategy as previously mentioned and illustrated in Figure 6.9. We use random barycentric coordinates described in Section 6.1.4 to generate our starting point for the ray. Note that in step 8, if neither hitsA or hitsB has a value of one, then either: *a)* the triangle is positioned in a way that its rays hit another part of the surface (e.g. with a scanned upright human, the rays from one ankle could genuinely hit triangles in the opposite leg) or b) we are dealing with a triangle that is inside the model, in the context of laser surface scans, this would typically be a self intersection of the fitted mesh. Since it is not possible to distinguish between the two cases, we choose another triangle from the same group, ultimately our search for a reliable one hit ray allows us to cope with these degeneracies. We present results on scanned models in Section 6.1.5.

6.1.3.5 Robust ray tests

The previous section describes how a ray is tested for intersection with all the triangles in the model. If degenerate [8] triangles are present in the model, ray-triangle intersection routines typically reject the triangles from consideration as small triangles can cause numerical problems when finding the coordinates of the intersection point. For instance [Sun01] finds barycentric coordinates building on intersection methods ([Bad90], [MT97]) and performs a cross product check on two of the triangle's edges, rejecting the triangle if the cross product is a zero vector. Unfortunately by doing so, we could potentially be ignoring a triangle hit count, and incorrectly invert or not the surface. The likelihood of a ray intersecting a degenerate triangle increases if the population of very small triangles is large (Table 6.1 reveals that 21% of triangles in the Happy model and 15.7% of triangles in the Dragon model have an area smaller or equal to 1e-8. A plot of all triangle areas can be seen in Figure 6.23. This likelihood also increases if the small triangles are evenly distributed across a model- Figure 6.24, shows triangles colour coded

[8] triangles that have a very small area such as that of thin triangles or very small triangles.

according to their aspect ratio, using our colour mapping technique detailed in Figure 4.8 and Gueziec's triangle fairness formula [Gue96], where an aspect ratio of 1 (blue) represents near equilateral triangles, and 0 (red) thin/*degenerate* triangles.

Model name	Model radius	# vertices	# triangles	# edges	average edge length	average triangle area	# triangles area<=1e-8	# triangles area<=1e-9
Bunny	0.116616	35,947	69,451	104,288	1.4e-3	8.2e-7	0	0
Blade	357.223	882,954	1,765,388	2,648,082	1.33114	0.603532	0	0
Happy	0.112664	543,652	1,087,716	1,631,574	3.4e-4	5.14e-8	230064	73453
Dragon	0.112592	437,645	871,414	1,309,256	4.5e-4	8.33e-8	137409	43786

Table 6.1: Model resolution and number of potential numerically problematic triangles in the two rightmost columns.

We found a fast and reliable solution for discarding degenerate triangles from our test. Recently Xu et al. [ZZL03] noted that a ray can be represented as the intersection of two non-parallel planes. For a triangle to intersect the ray it needs to intersect both planes, to intersect a plane the triangle's vertices need to exist in both the positive and negative half-spaces of the plane. By making one plane pass through the origin, the intersection test amounts to three distance measurements where each of the triangle's vertices is measured for distance against the plane. The method is particularly fast because there is no d component in the plane equation and no vector normalization is required, since only the sign of the distance is needed (Figure 6.22). Most triangles in a model, will be immediately rejected from consideration with the first plane. If triangles pass the test, then a second plane is built passing through the origin of the ray and perpendicular to the first plane, again no normalization is required. Finally, only if the ray intersects the second plane is the intersection point calculated with the intersection algorithm of choice.

Although degenerate triangles are not mentioned in their paper, we found the *three distance measurements* to be quite reliable in rejecting degenerate triangles that are not even close to the ray.

The computation(s) saved by only intersecting against triangles of the same normal group and through the use of the intersecting planes technique, in which most triangles are rejected by testing only the first plane, means that we can address large models without the use of a space partition data structure. These time savings (Table 6.2 versus Table 6.3) also allow us to further strengthen the reliability of our ray test by always firing a retro ray and only making a decision

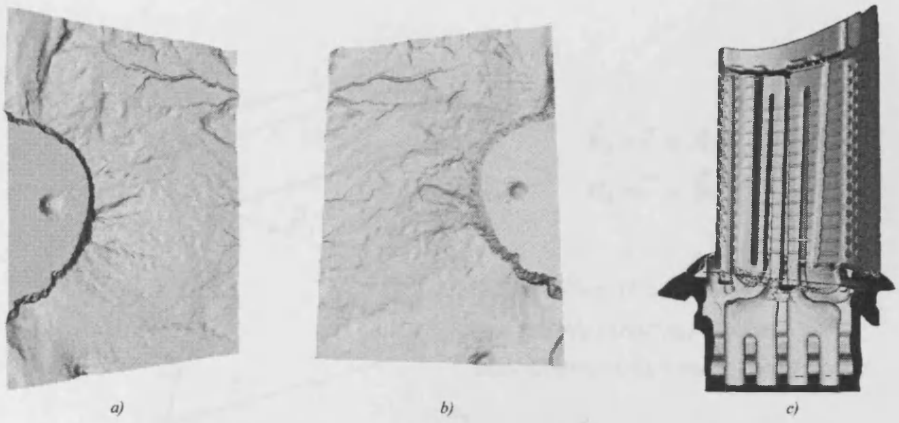


Figure 6.21: *a)* and *b)* Crater lake: open surface direction ambiguity.
Valleys, mountains and crater in b) could respectively be ridges, depressions and a plateau in a).
c) shows how an option to quickly invert normals might be useful for visual inspection of interior details.

on an object's orientation if only one of the rays reports a one triangle hit count, not both. If both rays report a one triangle hit, then we are in the presence of a dangling triangle attached to the object or a conventional open surface.

As mentioned at the end of Section 6.1.3.1, our algorithm has a different mode for when the user indicates that the model is an open surface. In this mode holes are not triangulated, and accordingly the ray test is adjusted for surfaces, where the constraint of only having a one hit with one of the rays does not apply. Indeed in this mode no retro rays are fired if the forward ray only hits one triangle.

The orientation of open surfaces is arbitrary, there is no automatic way of establishing the direction in which the surface is supposed to be oriented and ray tests will mostly report a one triangle hit on either direction/side. Consequently, a computer algorithm has a 50% chance of getting the orientation right. To address this problem, we have added an additional parameter that allows the user to simply invert all the triangle normals in the model, this can be done in 7 seconds on the Turbine blade model with a G4 500Mhz power book, and can be quite useful for temporarily visualising the interior of 3D models (Figure 6.21 shows how previous front facing triangles that were obscuring interior details become back facing and culled after normal inversion).

Timings for the large models with multiple surfaces are reported in the rightmost column, and in the last three rows of Table 6.3. Timings for the repair operation of non-manifold edges

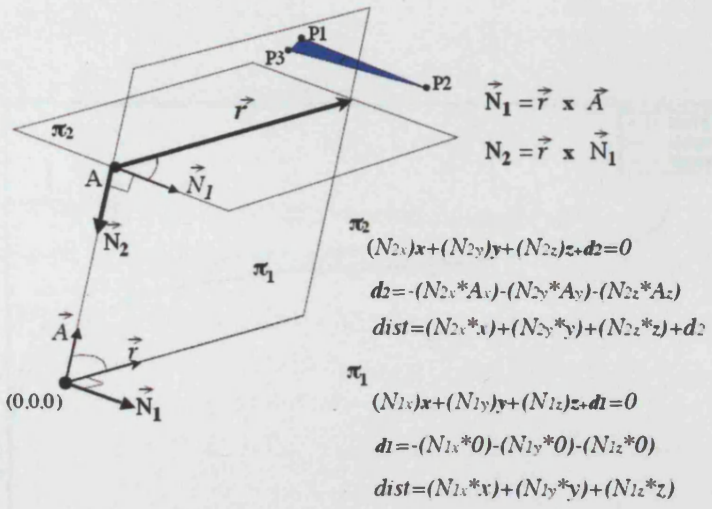


Figure 6.22: Xu's ray represented as two planes.

and vertices can be found in Table 6.4.

Degenerate thin triangles or creases are well known to cause rendering artifacts, as illustrated in Figure 6.25 *a)*. To cure this problem, we can create a weighted triangle normal. Instead of adding all triangle normals around a vertex and dividing the normal by the number of triangles, we calculate the sum of triangle areas around the vertex, multiply each triangle normal by their area divided by the total area, and then add it to the vertex normal. This way, thin triangles with small areas have almost a zero contribution to the vertex normal. Figure 6.25 shows results of area weighting triangle normals with Gouraud shading in *b)* and flat shading in *c)*.

6.1.4 Implementation issues

Often the reliability of a given approach can be affected/conditioned by the implementation choices taken. Since our rays need to be reliable, in this section we provide some implementations details of the following components:

- Computing a ray's initial position
- Determining inside/outside in ray-triangle intersection
- Normal grouping for large models, iterative solution
- Edge representation

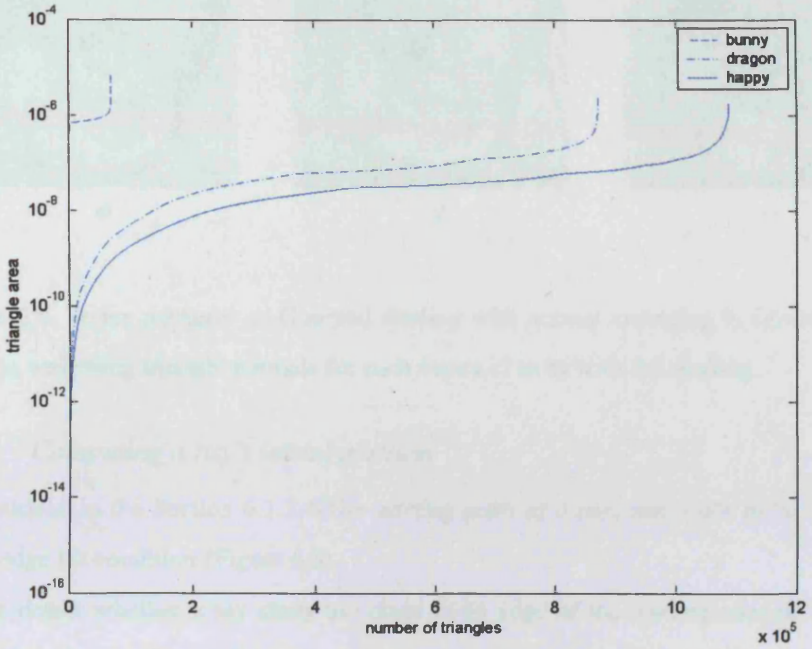


Figure 6.23: *Triangle areas:* 21% of the triangles in the Happy statue model and 15.7% of triangles in the Stanford Dragon model have areas of a floating value below 1e-8.

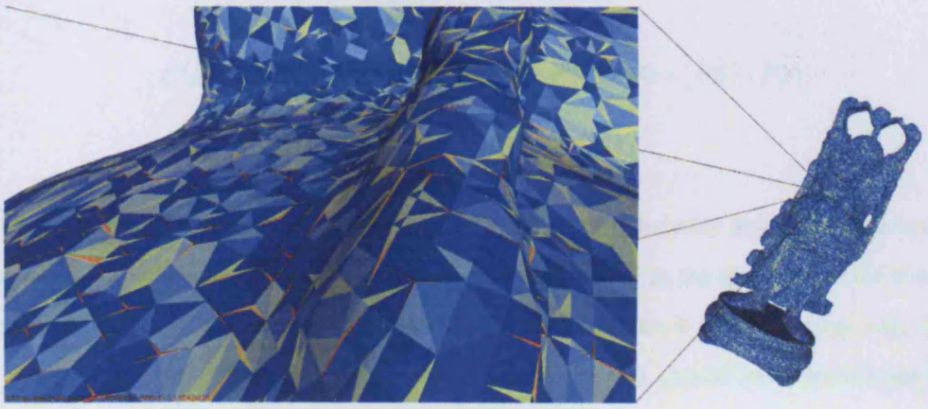


Figure 6.24: Small triangles distribution on the Happy statue model.

a) b) c)

Figure 6.25: Vertex normals: *a)* Gouraud shading with normal averaging *b)* Gouraud shading with area weighting triangle normals for each vertex *c)* as b) with flat shading.

6.1.4.1 *Computing a ray's initial position*

As mentioned in the Section 6.1.3.4/*The starting point of a ray*, one needs to be careful with the ray edge hit condition (Figure 6.9).

We detect whether a ray starts too close to an edge of the starting triangle by forming two vectors $v1$ and $v2$ with the randomly created starting point and two vertices of the triangle (Figure 6.26 *a)*). If either angle $\partial 1$, $\partial 2$ or $\partial 3$ formed between one of these vectors and an edge is smaller than or equal to half a degree we classify the ray as being on the edge, it is discarded and a new randomly starting ray for the triangle is spawned. Figure 6.26-*a)* illustrates the two vectors $v1$ and $v2$, and how an initial random starting point is computed. We call a pseudo-random[9] number generator [Lan03] three times, and divide each number by the sum of the three. A starting point can then be calculated by:

$$P(\beta 1, \beta 2, \beta 3) = P1 + \beta 2 * (P2 - P1) + \beta 3 * (P3 - P1)) \tag{6.4}$$

$$where\, \beta 1 + \beta 2 + \beta 3 = 1 \tag{6.5}$$

once the starting point is clear of the edges, we make sure numerically that the ray intersects the starting triangle. For the retro ray, we move the starting point in the direction of the triangle's normal to the front of the triangle by an offset equal to one tenth of the average edge length of the model. This offset is therefore model dependent and should not intersect any other triangle, if it does then our intersection count reliability criteria of 1 intersection (described in section 6.1.3.4) will force a ray to be fired from another location. For the forward ray, we move

[9] given an initial seed/starting number, a pseudo-random number generator creates the same sequence of numbers for that seed everytime it is run, hence the word pseudo.

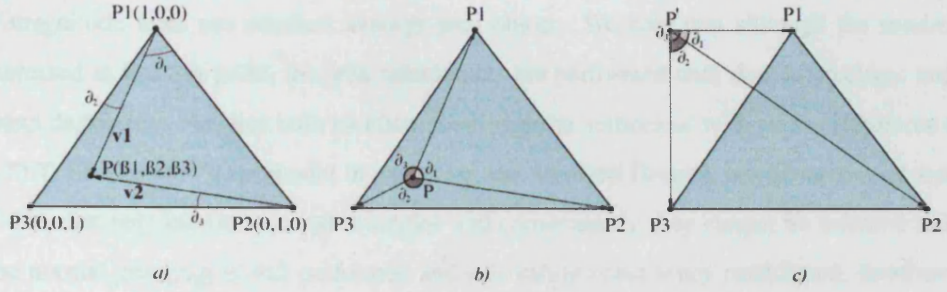


Figure 6.26: *a)* Barycentric coordinates of ray start point with edge nearness angle tolerance tested on angles $\partial 1$, $\partial 2$ and $\partial 3$

b) ray-triangle intersection, points inside triangle will have $\partial 1 + \partial 2 + \partial 3 = 360^o$

c) ray-triangle intersection, points outside triangle will have $\partial 1 + \partial 2 + \partial 3 < 360^o$, in picture $\partial 1 + \partial 1 + \partial 2 + \partial 3 = 180^o$

the starting point behind the triangle with the inverse direction of the triangle's normal by the same amount. A zero offset, whilst mathematically correct, gives unreliable numerical results. Similarly, although rays are infinite, in practice an end point for the ray is used. Defining a ray's position and direction by specifying that the second point is offset from the start point by 3 times the object's radius in the direction of the triangle's normal. This offset ensures that the ray will be tested for intersection with all triangles of the object. If the object's radius is smaller than 1, we default this number to 10, as in practice this has also proved to be a problem. Likewise, if the average edge length of the model is larger than 1, we can adjust the average edge length used in the above offset to 1.

6.1.4.2 *Determining inside/outside in ray-triangle intersection*

In Section 6.1.3.5 we showed a fast ray-triangle intersection rejection test that copes with degenerate triangles, we have also shown how to compute the intersection point in Section 6.1.3.4. The last test for the intersection is to determine whether the computed intersection point is inside or outside of a given triangle. We have adopted the test described by [Bou97], where vectors are created between the triangle's vertices and the intersection point. If the sum of the angles between these vectors is 360^o (Figure 6.26-*b)*) then the ray is deemed to intersect the triangle, otherwise if the sum of the angles is less than 360^o by 1 degree in our implementation, then the ray is considered as not intersecting the triangle (Figure 6.26-*c)*).

Whilst the rejection test can cope with degenerate triangles, the intersection calculation is not immune to numerical problems. Instead of comparing the crossproduct of a triangle edge

to a zero vector [Sun01], we consider the triangle to be degenerate if the area falls by an order of magnitude from our smallest average area object. We note that although the models are expressed in floating point, the area calculations are performed with double precision and can detect degenerate triangles with identical floating point vertices or with value differences close to FLT_EPSILON.[10] One model in particular, the Stanford Dragon, has 32 surface groups out of 151 that only consist of small triangles and consequently they cannot be oriented reliably. The normal grouping is still performed and orientation consistency established, however rays cannot be fired and the group is skipped after all triangles in the group are attempted. Finally, we would like to point out that in these cases, the problematic surface groups were surfaces that have arbitrary orientation in the first place.

We have tested the logical outcome/triangle hit count of [Sun01] and [Bou97] algorithms with an area tolerance of $1e^{-9}$. The test consisted of forcing rays to be fired from every triangle in a group against all other triangles, and is $O(N^2)$ per surface group. The operation was only completely verifiable as identical with small models, large models with small average areas had identical triangle hit counts for at least a few thousand of the triangles tested. The inside/outside method was chosen based on the simplicity of the calculations, and hence stability.

6.1.4.3 *Normal grouping for large models, iterative solution*

As mentioned in Section 6.1.3.3, the recursion depth whilst normal grouping can be quite deep with large models, and will break/exceed the computer's stack memory limit. To address this problem we create in the heap a temporary stack/variable large enough to store the function arguments of all the would be recursion calls. By loading these arguments in inverse order onto the stack, and always iteratively retrieving the last arguments from the stack, we can mimic faithfully the program behaviour of the recursive version. The main difference being is that the portion/section of RAM[11]/main memory used is from the computer's heap memory rather than from the stack memory which is significantly limited in size.

[10] FLT_EPSILON is the number/constant set by the compiler that represents the smallest increment ϵ that can be added to a float x, such that $x + \epsilon! = x$ on a particular machine, in most machines this number/precision is $1.1e^{-7}$.

[11] RAM: random access memory

Algorithm 11 Pseudo-code of normal grouping, *iterative solution.*

```
template<class F>
struct fixstate
{ F *f; int fgroup;
}
template<class P,class F, class E> void GOb-
ject<P,F,E>::fixvertexnormals(face, fgroup)
    // define a stack element that holds our type of faces/triangles
    typedef fixstate<F> stackelement
    // declare&create a stack to hold the maximum earlier recursion depth
    vector<stackelement> fixstack // STL vector/array
    fixstack.reserve(totalFaces) // do not wish to resize all the time
    // temporary stack element el, holds current function args
    stackelement el
    el.f=face
    el.fgroup=fgroup
    // load stack with current state element
    fixstack.push_back(el)
    while(fixstack.size()!=0)
        // get latest addition to stack, load args stored in stack element
        el=fixstack.back()
        face=el.f
        fgroup=el.fgroup
        // reduce size of stack
        fixstack.pop_back()
        // as recursive version_use function args
        face-> wt = fgroup
        v1id = face->firstvertex()  v2id = face->secondvertex()  v3id = face-
        >thirdvertex()
        nf1=getadjacentfaceatedge(v1id, v2id, face)
        forcevertexorder(face, nf1)
        nf2=getadjacentfaceatedge(v2id, v3id, face)
        forcevertexorder(face, nf2)
        nf3=getadjacentfaceatedge(v3id, v1id, face)
        forcevertexorder(face, nf3)
        // items are loaded in inverse order on stack, for later use
        last element added to stack is more recent equivelant recursed element
        if(nf3->get_groupid()==0)
            el.f=nf3 el.fgroup=fgroup
            fixstack.push_back(el)
        if(nf2->get_groupid()==0)
            el.f=nf2 el.fgroup=fgroup
            fixstack.push_back(el)
        if(nf1->get_groupid()==0)
            el.f=nf1 el.fgroup=fgroup
            fixstack.push_back(el)
```

6.1.4.4 *Edge representation*

In order to cope with multiple instances of the same edge in the model, we adopt the convention that edges are stored with their vertex references in ascending order and only once. If an edge has the same vertex references in reverse order, we can detect that it is identical to one already in memory and not store it. Only border edges are temporarily stored for the hole triangulation process. Storing all edges of a model can be memory expensive in the case of large models

(according to Euler's[12] formula, recall that Edges=Vertices+Faces-2 and Table 6.1, 5^{th}*column*) and unnecessary as edge connectivity queries can be done with the vertex connectivity information alone. Also, if the choice of language is C++, it is convenient to define logical *operators* <, >, <=,>=, ==, !=, so that the STL (standard template library) *sort* function can sort our lists of edges in the hole triangulation phase, without extra code being required.

6.1.5 Results

We have tested our algorithm with several objects of different types, for instance synthetic objects (Figures 6.27, 6.28), open surfaces (Figures 6.21, 6.28), laser scanned models containing several non-manifold configurations (Figures 6.29, 6.30, 6.31), and large models (Figure 6.32). We note that in these figures the illustrations corresponding to the original models will often have parts that appear to be missing. These parts are most likely to be incorrectly oriented to point towards the inside of the model. The rendering acceleration technique of back-face culling simply does not render the triangles of those parts. These parts become visible once our algorithm is applied to correct the orientation of all normal groups from the original models.

In this work we have used data from a Hamamatsu Body Lines scanner to produce the mannequin and Igor models. This scanner offers a 1-2 mm accuracy over approximately regular samples at 5 mm spacing over 400 horizontal slices of the body [Hor98]. We have also used a surface reconstruction software called Cocone, freely available at [Dey02], based on [ACDL02] to fit a surface to the scanned point cloud. The experiments were carried out on a PowerBookG4 500MHz, 1 Gbyte RAM.

Time savings from our optimized ray test (Table 6.3, rightmost column) can be compared with the unoptimized ray test (Table 6.2, rightmost column). With our unoptimized ray test, we can see that the dominant time factor was the number of mesh degeneracies present/number of surface groups. The time increases with how degenerate the model was, for instance Igor2 (fifth row of Table 6.4) has fewer triangles than Igor3 but more normal groups (third column) and takes longer. In practice there was no advantage in comparing these noisy surface groups with other triangles in the model, this fact together with the fast ray-triangle intersection/rejection test allowed us to tackle large models with several noisy surfaces.

The third column from the right in both tables shows how useful it was to use our two opposing rays strategy, as calculating the ray opposite to a triangle normal was determinant in finding the correct orientation of several of the normal groups. The timings for repairing non-manifold vertices and edges can be found in Table 6.4. The following figure presents an

[12]although this formula is only valid for closed surfaces without holes, we note that it is a useful indication of the number of edges on models that are not dominated by borders

Figure 6.27: *Left:* inconsistent normals, *right:* normals after applying our algorithm.

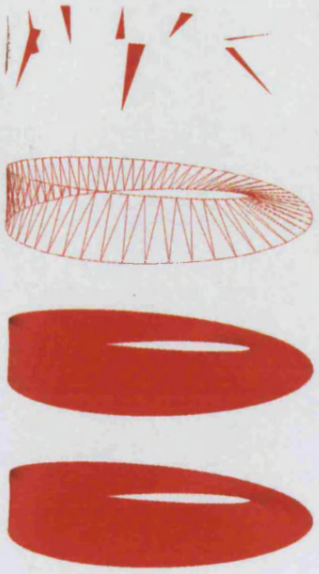


Figure 6.28: From top to bottom: Mobius strip with inconsistent normals, wireframe results after applying our algorithm, flat shading results, Gouraud shading results

inconsistent symmetrical object on the left and shows results after applying our algorithm on the right.

Figure 6.28 shows a Mobius strip with inconsistent normals (top), and results after applying our algorithm (below). The transition of the normals from outside to inside can clearly be seen. The object is completely front facing from this view, and completely invisible on the other side facing the viewer. A surface is said to be *orientable* if one can distinguish two different sides [Hof89]. Hoffmann [Hof89] gives as examples of non-orientable surfaces the Klein bottle and the mobius strip.

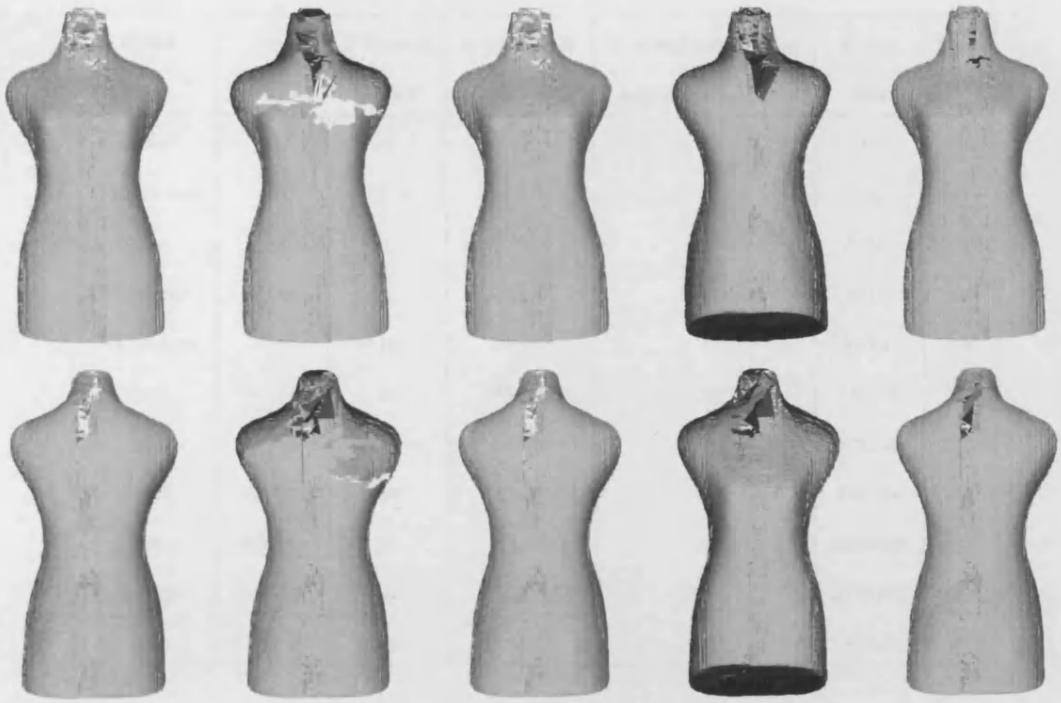


Figure 6.29: Mannequin (*top:* front, *bottom:* back) - from *left* to *right*: original model, Ivnorm[default], Ivnorm[counterclockwise], Ivnorm[clockwise], our result.

Model name	# triangles	# Normal Groups	# triangles in largest group	# ray starts on edge	# opposite orientation rays with single hits	# rays shot	Time (*s*)
Cube	12	1	12	1	0	12	<1
Mobius strip	120	1	120	1	1	240	<1
Mannequin	31,662	18	31,544	55	6	2,461,786	20
Igor1	66,164	44	65,806	89	20	8,703,944	69
Igor2	62,982	80	62,280	160	43	16,182,298	126
Igor3	68,590	58	67,977	116	26	11,445,967	91

Table 6.2: *Rightmost column:* timings as published in [OS02], where ray test is not optimized

Model name	# triangles	# Normal Groups	# triangles in largest group	# opposite orientation rays with single hits	# rays shot	Time (s)
Cube	12	1	12	0	12	<1
Mobius strip	120	1	120	1	120	<1
Cow	5,804	1	5,804	0	23,216	<1
Dinosaur	28,064	2	28,060	0	56,152	1.02
Mannequin	31,662	18	31,544	4	63,500	1.4
Igor1	66,164	44	65,806	10	132,152	3.5
Igor2	62,982	80	62,280	20	125,746	5.2
Igor3	68,590	58	67,977	10	136,734	4.3
Dragon	871,414	151	874,386	0	3,510,829	47.8
Happy	1,087,716	1	1,087,716	0	2,175,432	34.5
Blade	1,765,388	295	1,760,072	41	3,632,506	138.7

Table 6.3: *Rightmost column:* timings with optimized ray test and large models/bottom three *rows*

Model name	# triangles	# Normal Groups	# triangles in largest group	# opposite orientation rays with single hits	# nm_edges/ nm_vertices	# rays shot	Time(s) (rnm_e/fix/rnm_v)
Cow	5,804	1	5,804	0	0/1	23,216	0.05+0.2+0.1
Dinosaur	28,064	2	28,060	0	0/2	56,152	0.2+0.8+0.5
Mannequin	31,662	50	31,544	5	32/61	63,480	0.2+2.5+0.7
Igor1	66,164	176	65,806	10	132/239	13,404	0.45+9.6+1.3
Igor2	62,982	231	62,280	20	152/261	126,096	0.4+12.3+1.2
Igor3	68,590	337	67,977	11	284/465	136,734	0.4+17.4+1.4

Table 6.4: *Rightmost column:* timings for repair on nonmanifold edges (rnme), fixing normals (fix), and repair of non-manifold vertices (rnm_v)
third column from right: number of non-manifold edges (nm_edges) and non-manifold vertices (nm_vertices)

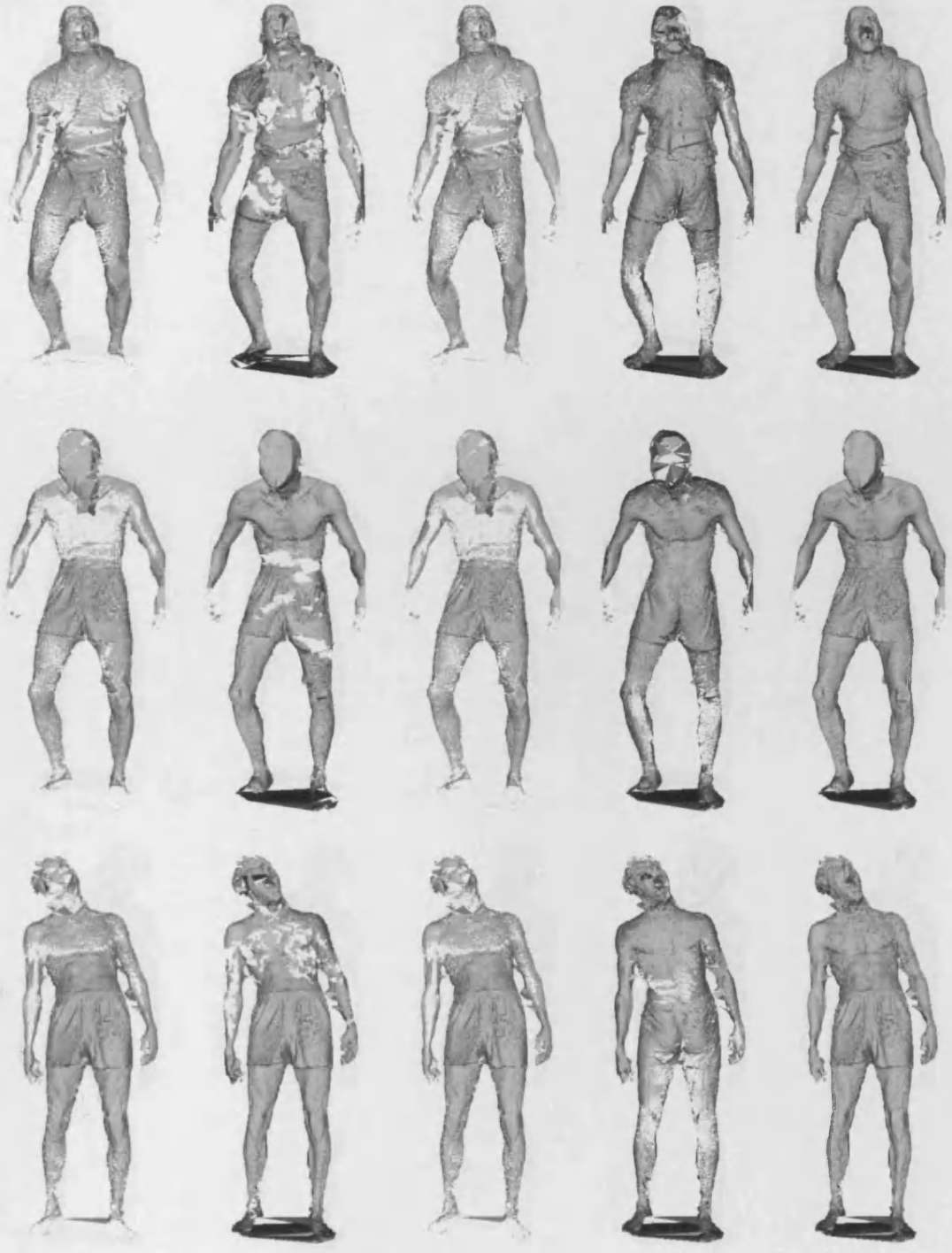


Figure 6.30: Igor 1, 2, 3 (front) - from *left* to *right*: original model, Ivnorm[default], Ivnorm[counterclockwise], Ivnorm[clockwise], our result.

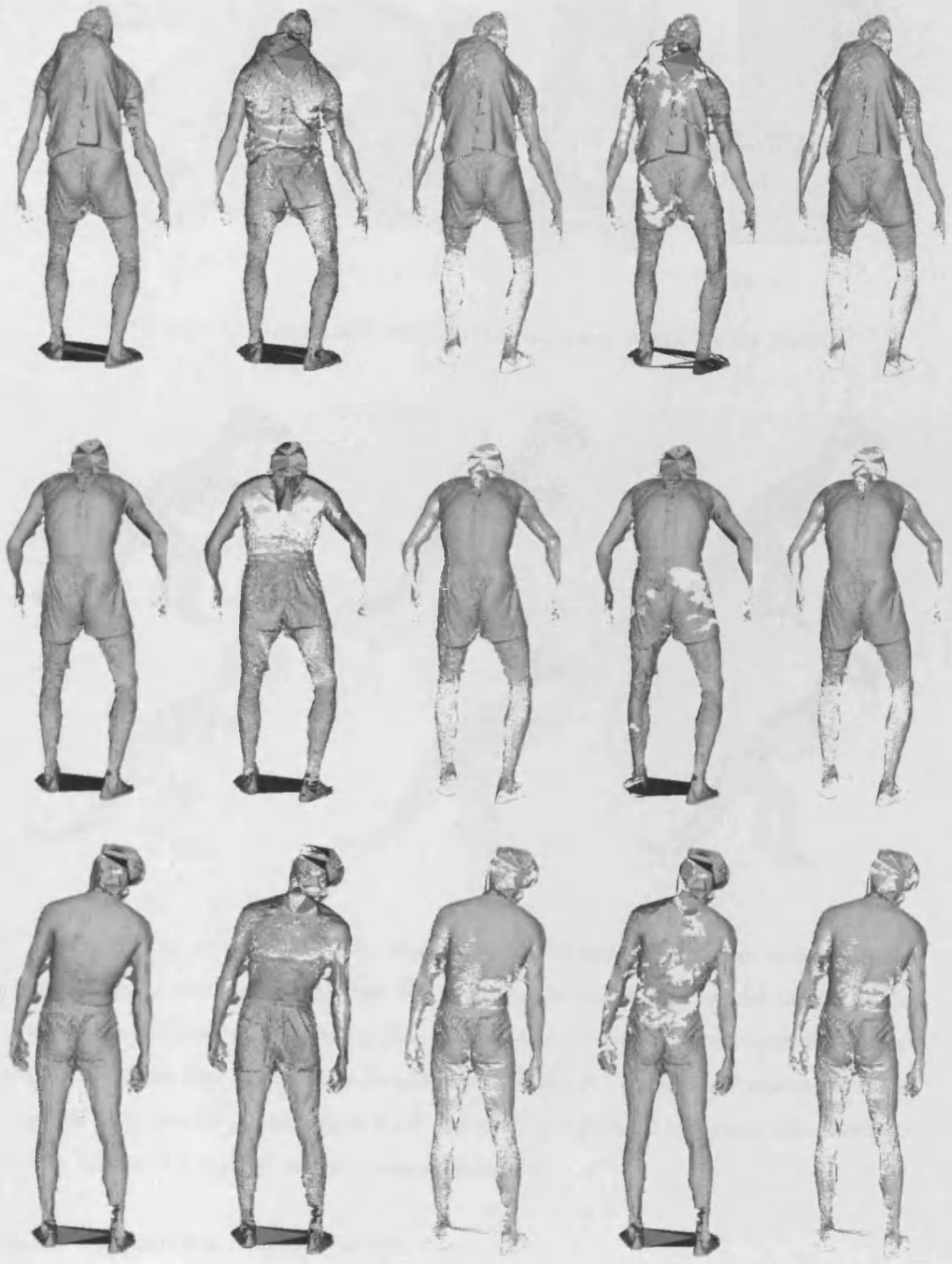


Figure 6.31: Igor 1, 2, 3 (back) - from *left* to *right*: our result, Ivnorm[clockwise], Ivnorm[counterclockwise], Ivnorm[default], original model

Figure 6.32: Results with Stanford Dragon, Happy statue, Turbine Blade.

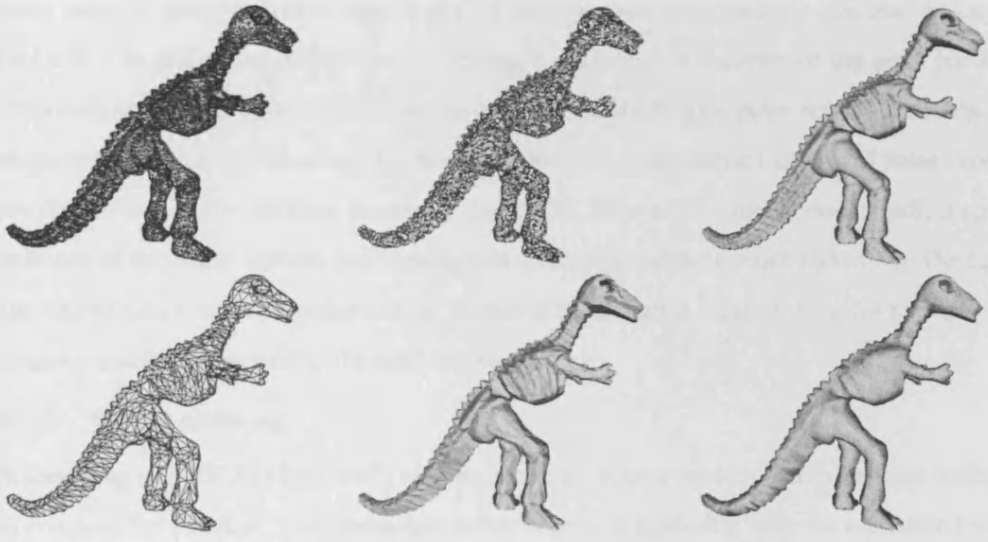


Figure 6.33: **Top row:** *left:* dinosaur model (28,064 triangles) in wireframe, *middle:* original model with inconsistent normals and two non-manifold vertices in Gouraud shading, *right:* dinosaur model in Gouraud shading after applying our algorithm with non-manifold vertices repaired **bottom row:** result from Simplification Envelopes (SE) with our repaired model of top row right and SE parameters -E 0.2% of bounding box (3,682 triangles), *left:* wireframe result, *middle:* flat shading, *right:* Gouraud shading.

6.1.6 Application to level of detail

As mentioned in Section 6.1.1.1, inconsistency in triangle normals direction can be a problem in simplification algorithms and other geometric processing tools. For example, [CVM+96] requires that the object is consistently oriented and has no non-manifold configurations. Fig-

ure 6.33 shows the dinosaur model distributed with the cocone software package [Dey02]. The model has inconsistent normals and two non-manifold vertices. The figure shows simplification results that were possible by using our algorithm for establishing orientation consistency and repair of non-manifold vertices.

6.2 Vertex cleaning of 3D models

This section briefly presents the problem of duplicate vertices in the context of level of detail algorithms in Section 6.2.1, and presents a simple solution in Section 6.2.2.

6.2.1 The impact of duplicate vertices in level of detail algorithms

The problem of duplicate vertices is often due to bad attribute management. For example, in the context of a hierarchical CAD part such as a cylindrical pipe which has associated with it some transformation matrix to place it exactly adjacent to another knee or t-junction object, two problems can arise: one is that when exporting the model two vertices for the same point are produced; the other is that rotation matrices can accumulate floating point errors making the two points non-identical in 3D space. The second problem worsens when the level of noise exceeds the dimensions of the smallest feature in the model. Figure 6.34 shows cracks indicating the presence of duplicate vertices and making LoDs unusable without much reduction. The figure also shows that a lot of triangles can be deleted if the model is cleaned with, for example, our simple algorithm presented in the next section.

6.2.2 Vertex cleaning

Schmalstieg's LODESTAR [Sch97] uses an octree to retrieve vertices inside the same leafnode to compare for deletion. This technique is fast as only neighboring cells are considered when deleting a vertex, instead of the all the vertices of the entire model. However, we believe that there is a type of noise in the coordinates of vertices of 3D models that could benefit from the following consideration. If a vertex A is within an euclidean distance tolerance of a vertex B, and B is within a tolerance of a vertex C, it would be desirable that only one of the three vertices remained. But, if care is not taken during the deletion process, B can be deleted and C is then not deleted by A. We propose the strategy outlined in Algorithm 12 to cater for this situation. Namely we allow vertices that are deleted by other vertices to still delete other vertices, the algorithm has four parts, an initilization part, a marking/comparison part, a simplification part, and the actual deletion part.

In part1 we use two fields in each vertex, one field ColX keeps the vertex's id or name, all vertices are assigned their corresponding id at the start. The second field ColZ which is initially marked as zero in all vertices keeps track of whether a vertex is deleted by another or not, a

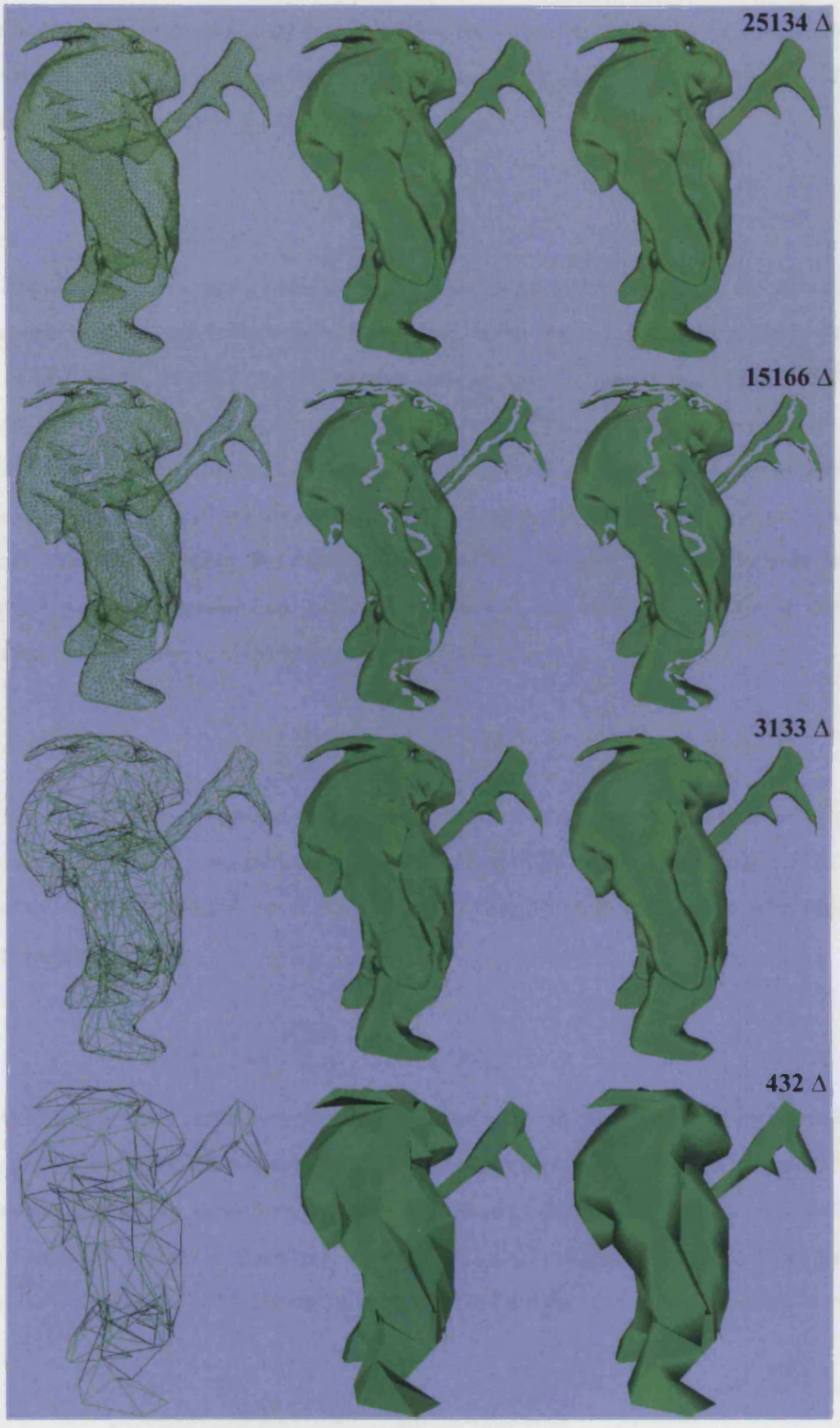


Figure 6.34: Cracks in *LoDs* due to duplicate vertices - top row: *left:* wireframe rendering of the original model, *middle:* flat shading, *right:* Gouraud shading; second row from top: cracks appear where duplicate vertices were simplified in different directions; third and fourth row: simplified results after using our vertex cleaning algorithm.

non-zero positive number indicates the vertex id of the vertex that deletes it. We term the vertex that is deleted as *deletee*, and the vertex that deletes a *deleter*, deleters are also vertices that are not deleted by others and have a ColZ of zero.

In part2 we form vectors with each vertex p and every other vertex p2 in the vertex array of the model, and calculate the length of the vector. If this length is below for example a tolerance of FLT_EPSILON (the smallest number one can add to a number that makes it different in a given machine), then the following considerations are made. If p has not been marked for deletion, the function set_all_parentids marks p2's ColZ with the id of p, it also retrieves the other vertices that may delete p2 and marks them with the id of p. Similarly the function get_last_parentid retrieves the id of the vertex that deletes another but is not deleted by others. In order to distinguish whether a vertex has been marked for deletion by vertex zero, or whether the vertex is unmarked, one is added to ColZ when marking for deletion.

If p is already to be deleted we check whether p2 is also already to be deleted, if p2 is not the same vertex as p we mark p2's ColZ with the id of the vertex that deletes p. If p2 was unmarked, again we check to see if the vertex is not deleting itself and mark p2 with the id of the vertex that deletes p.

Before any vertex is deleted, we resolve and simplify all transitive relations in part3 (resolve hops). Specifically we update the ColZ field of vertices that are to be deleted with the id of vertices that are not themselves deleted, this part traces back deleter vertices until finding a vertex that is not deleted. For example, a vertex B that is to be deleted by a vertex A can delete a vertex C. Here the ColZ field of vertex C is updated to have the id of vertex A instead of vertex B.

Finally in part4, any triangles around a deletee are transfered to its deleter. The transferred triangles are updated to have the id of the new vertex.

Algorithm 12 Pseudo-code for duplicate vertex deletion

```
function clean_duplicate_vertices () {
for(i=0 i<number_of_vertices i++) //PART1-initialization
  p=atVertexArray(i)
  p->set_ColX(i) // record name of vertex
  p->set_ColZ(0) // a positive tag, indicates the name of a vertex that deletes it
for(i=0 i<number_of_vertices i++) {//PART2-compare vertices
  p=atVertexArray(i)
  if(p!=0){
     vp1= get_last_parentid(p)
     for(u=0 u<number_of_vertices u++)
     { p2=atVertexArray(u)
      if((p2==0)||(p->get_ColX()==p2->get_ColX())){;}
      else {
      v=p2-p1
      length=square_root(v.X()xv.X() + v.Y()xv.Y() +v.Z()xv.Z())
      if(length<=tolerance) { //duplicate vertex!
          vp2= get_last_parentid(p2)
          if(vp1>0) //p is deletee, allow it to delete p2s
              if(vp2>0)//p2 is already deletee
                if(vp1!=vp2)//self-delete?
                  set_all_parentids(p2, p->get_ColZ())
              else //self-delete?
                if((vp1-1!=p2->get_ColX())
                  set_all_parentids(p2, p->get_ColZ())
          else { //p is deleter,mark
              set_all_parentids(p2, p->get_ColX()+1)}
     } }
} } }
for(i=0 i<number_of_vertices i++) //PART3-resolve hops
     p=atVertexArray(i)
     if(p!=0)
          if(p->get_ColZ()>0) //vertex is deleted by another
              vfinal=1; gotoid=p->get_ColZ()-1
              while(vfinal>0) {
               p2=atVertexArray(gotoid); vfinal=p2->get_ColZ()
               if(vfinal>0) gotoid=vfinal-1}
              p->set_ColZ(gotoid+1)
for(j=0 j<number_of_vertices j++) //PART4-delete marked vertices
     p2=atVertexArray(j)
     if(p2!=0)
          i=p2->get_ColZ()
          if(i>0) //vertex is deleted by another
              i=i-1 p=atVertexArray(i)
              if(p!=0)
               add faces around p2 to p1
               rename vertices of these faces to have vid i
               instead of j
               deleteVertexKeepConnectivity(j)
              else deleteVertexKeepConnectivity(j)
```

We note that the complexity of the algorithm presented is O(N*N), the two nested for loops of part2 can be easily replaced by accessing only the vertices within a node of an octree containing a given vertex, and comparing them only with the vertices of their adjacent nodes. In the next section we present some results.

6.2.3 Results

Figure 6.35 shows that a significant reduction was possible with a hierarchical CAD model without much visual difference. When duplicate vertices are deleted, a mesh surface stretches across objects rather than tearing the objects apart. The level of reduction can be explained by the large number of curved objects representing the valve handles.

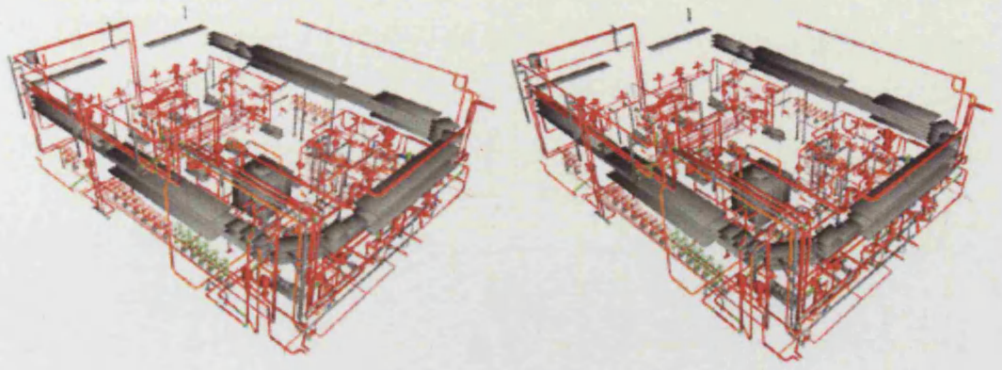


Figure 6.35: Simplification after duplicate vertex cleaning: *left:* original thermal power plant model 545,452 triangles; *right:* 245,452 triangle simplified model after using our vertex cleaning algorithm and simplification.

In order to fully test the correctness of our cleaning algorithm, we used the 8 times subdivided icosahedron from Section 3.5.3.1, that creates 1,310,720 triangles and 1,310,712 vertices on a unit sphere. The subdivision process does not check for adjacent vertices when creating new vertices, hence several duplicate vertices are created in every direction on the sphere. One side effect of the creation of duplicate vertices is the creation of border edges along vertices that were duplicated. Thus a simple test for the presence of border edges can be used to determine if all duplicate vertices were deleted. The face marking scheme illustrated in Figure 6.11 can be used to detect borders, a border edge will only have one triangle with a mark of two, instead of two triangles with a mark of two. Initially the sphere had 327,672 border edges, after using our cleaning algorithm the sphere had 1,310,720 triangles, 655,362 vertices and 0 border edges. For reference purposes using a 500 MGhz G4 desktop machine it took 982,513 seconds to process, or approximately 11 days (982,513s/ 86,400s). On an equivelant 2 Ghz machine today it would take approximately 3 days. However, if one were to only search vertices within the same volume and immediately adjacent volumes, the complexity of the algorithm would be reduced to O(N). Depending on the size of the volumes used in the search, the adjusted algorithm could

take minutes or less using small volumes in either machine, or again days if the volume were to be of the same size as the root of the octree. In the latter case, the complexity of the algorithm would again be O(N*N).

In the next chapter we present conclusions and future work.

Chapter 7

Conclusions & Future work

This chapter concludes our work on the Octree Interaction Engine, the system we developed in order to implement our approach to vertex classification for non-uniform reduction. We summarize each chapter and draw conclusions in Section 7.1. We assess the claims of our main hypothesis in Section 7.1.1, our sub hypothesis in Section 7.1.2 and Section 7.1.3 and our secondary hypothesis in Section 7.1.4. We discuss and list our contributions in Section 7.1.5. A summary of the thesis is given in Section 7.2. Finally we present ideas in progress for extending the system in Section 7.3.

7.1 Conclusions

The literature review presented in Chapter 2 provided both a historical and systematic account of the evolution of a number of different components of interactive visualization systems. These components were developed in response to a variety of challenges that emerged with the increase in size of 3D models and their greater availabilty. Such components helped us design a new efficient system for editing a range of large models that can reside in main memory today without the lags associated with visualizing LoD content from secondary memory. Through several independent investigations, such as GOLD [BGB+05], HLoD [EM00], Far-Voxel [GM05] and our own work [OB05] it became apparent that it was not possible to perform mesh refinements at the triangle level as in early systems fast enough in order to meet the requirements of a visualization system, in particular, during large changes of view direction when viewing large models. Variations in frame rate have been shown to affect the performance of tasks such as grasping an object in VR [WSNR96]. Newer visualization systems switched between complete LoD surfaces instead, or entire complex CAD parts were switched by alternate simpler HLoDs. However, as we described in Section 2.5.2 lags in displaying new content inherent from accessing secondary memory are still present in such systems.

In Chapter 3 we considered the context of viewing and intensive editing of large scanned

models and went a step further, by eliminating LoDs and their associated memory overheads that would, for such models, force the visualisation system to be bound to secondary memory. Furthermore we presented PNORMS in Section 3.5 a novel normal and colour attribute compression algorithm that allowed us to reduce the total memory of models by 44% with a maximum error of 1.3 degrees in representation of the normals, enabling us to load even larger models into main memory. We showed that a five-times subdivided icosahedron produced approximately two and a half times more normals for the encoding than a five times subdivided octahedron as widely used in the literature. The greater number of normals gives developers the option to use byte aligned addresses that do not require decompression, thereby allowing the graphics cards to perform other tasks. We showed that it was possible to obtain sub degree accuracy with the larger normal databases of representative normals obtained by subdivision of the icosahedron and that subdivision of the icosahedron also produced less mean and maximum representation errors than did subdivision of other platonic solids.

With a laptop machine equipped with 1 GByte of main memory we loaded and were able to interact with a 28 million triangle model. With today's equivelant laptops equipped with, for example, 2 GBytes of main memory one can load models of 56 million triangles (Figure 7.3; b and f). Furthermore, some desktop machines are equipped with 4 GBytes of main memory which would allow us to load models of over 100 million triangles. Such models are larger than several scanned models of Michaelangelo statues in the Stanford repository. Thus, models that were previously deemed out of core can now move in-core, and a system that can manage such models in-core can benefit from the speed benefits of main memory such as immediate access to parts of a mesh thereby accelerating their inspection.

We created a system that combined a compact octree and a coarse navigation skin with a single pass depth buffer strategy that ensured that the fine resolution geometry retrieved by the octree was never obscured by the navigation skin. The navigation skin, which is a clustered version of the model computed on the fly, provides an overview of the extent and general shape of the model. We developed a load balacing strategy based on pointers to hierarchical volumes of the octree which we termed the RenderArray.

Use of the RenderArray allowed us to prioritise the rendering of triangles of octree nodes close to the intersection point of the line of sight and the model and to adapt the available processing resources to the size of the loaded model. In particular, we showed that it was possible to use our sytem to render at approximately the same speed models of any size by creating a navigation skin of similar complexity, rendering the same triangle budget/size portion of the original model where the user is gazing and by choosing a RenderArray size that can

be processed robustly by the hardware for each model. For example, it was possible with a Powerbook G4 500 Mhz equipped with an ATI RageMobility128 graphics card to render with a window size of 600x600 at above 14 fps from any viewing direction a model of 10 million triangles and the same was possible with a 28 million triangle model.

Unlike visualization systems that use an octree to retrieve parts of the fine resolution mesh in blocks, our visualization system provides seamless and immediate access across blocks. The in-place sorting performed when creating our octree also made meshes more coherent and compared well with algorithms that prepare models for mesh streaming (Section 3.1.7).

We extended our system to work with textured, scanned models (Section 3.3) so that it would compute without manual intervention the new textures required for smaller view volumes. We overcame the problem of texture over-binding when too many texture switches between volumes take place, thus allowing one to visualise textured models.

We provided a solution for viewing objects with multiple surfaces, for example, one inside the other, and for viewing CAD models. Our solution enabled us to achieve one of the initial challenges that motivated this work - performing an extensive edit to the 596,872 triangle model of the human brain to produce a symmetrical, right brain hemi-sphere to match the left-hemisphere provided. We note that the realisation of this challenge supports our main hypothesis that it is possible to create a system that allows users to interact with models of any size that can be accommodated in main memory and mark features for the purposes of non-uniform reduction. However, prior to focusing on non-uniform geometry reduction we addressed problems encountered in the development of systems for uniform reducion of geometry. In Chapter 4 we presented a simple framework for incorporating mesh quality constraints such as the prevention of very thin triangles and the detection of mesh fold-over into an edge-collapse based simplification system. We studied the problem of ensuring that no "magic" numbers are used in the numerical solvers of systems based on quadric error representations. In particular, we presented an automatic condition number calibration method that chooses a tolerance appropriate to the model loaded. We showed that the units in which a model is expressed can inadvertently make systems appear ill- conditioned (see section 4.3.3) that thus lead to suboptimal results. For example, triangles with tiny edge lengths have areas of even smaller numerical value, which when used to weight the contributions to a quadric error, can make a system that would otherwise have healthy condition numbers look-ill conditioned. With our calibration system one need not scale a model or guess a scale factor.

Lindstrom et al. [LT99] showed that it was possible to achieve better geometric errors than obtained in Garland's QSlim [GS97] by not accumulating quadrics, and by optimising his

own quadratic functions for qualities such as volume preservation, triangle shape optimisation and volume optimisation. In Section 4.7 we made a direct comparison between approaches in which quadrics were accumulated and those in which they were not, but recomputed on the fly without accumulation. We used the same quadratic cost function as used in QSlim, and found that indeed the memory-less approach can produce similar or better geometric errors than QSlim does and, of course, without a significant memory overhead and was approximately only 13% slower than QSlim. In Section 4.5 we presented the algebraic formulation needed to constrain solutions to a perpendicular plane at border edges. This enabled us to work with open surfaces and avoid the borders from receding.

In Chapter 5 we extended the framework for incorporating mesh quality constraints in a uniform system of geometry reduction to add constraints to support the non-uniform reduction of geometry. Users were allowed to select manually features as presented in Chapter 3. We showed how manually selected areas of semi-regular meshes could be preserved using our system with a variety of simplification algorithms. We demonstrated also how the system could be used to completely preserve borders and colour discontinuity curves. In addition we showed an example of non-uniform reduction of geometry in a computer animation system that preserved more vertices around the joint areas of body scans in order to increase the quality of deformation when the models move.

In Chapter 6 we addressed some problems that, for a wide range of models, compromise the reduction of their geometry. Thus, in Section 6.1 we presented an algorithm that consistently orients surface normals even in the presence of non-manifold triangles which are marked or deleted beforehand. In Section 6.2 we presented an algorithm that deletes duplicate vertices that would otherwise generate cracks in a model during reduction. Our two pass strategy ensures that all vertices within a set Euclidean tolerance from each vertex are deleted, whether they are strict duplicates or not. Finally in appendix A we present an animation system that computes skeletons of bodyscans automatically from surface landmarks. The joints of these bones were used to mark features for non-uniform geometry reduction.

7.1.1 Main Hypothesis #1

"It is possible, without the assumption that the model is small enough for interaction, to develop a system that enables one to interact with and designate 3D features on a model stored in main memory, for example, for non-uniform geometry reduction."

Our implementation of the Octree Interaction Engine supported the claim of our main hypothesis (see Section 3.1). Thus, for example, it enabled us to carry out an extensive edit to the human brain model. Approximately 4 hours of editing were required according to the edit-

ing procedure which was described in Section 3.1.6. The task involved manually marking and deleting approximately 25,000 triangles from grey and white matter surfaces of the 596,872 triangle model of the left hemisphere of the brain (see Figure 1.3, left). The task required both grey and white matter surfaces to be present simultaneously to ensure that the extrusion of the borders of the resulting hole cut so as to produce a "corpus callosum" joining the two halves that did not intersect the other surface. The final symmetrical model of both hemispheres was comprised of approximately 1.1 million triangles and can be seen being inspected with our system in Figure 3.5. We note that owing to the intricate nature of the grey and white matter surfaces, one could not resort to LoDs to preview the model as an excessive number of triangles would be required to guarantee non-intersection of the preview surfaces. Such intersections clutter the rendering and would hamper the editing task and slow down the rendering throughput.

Our system was also used to interact and mark feature areas for preservation in a non-uniform geometry reduction setting in Chapter 5. Our system allowed us easily to interact with the 126,108 triangle face scan model (see Figure 5.1) and to mark and preserve triangles to produce 10,000 triangle LoD models where 5,433 triangle where unmodified from the original model (see Figure 5.4).

We obtained similar rendering speeds of above 14 frames per second with a low end graphics card with models comprising 10 million triangles and 28 million triangles as reported in Figure 3.11. Visualisation of the latter model was made possible by use of our PNORMS compression algorithm (Section 3.5) and by the ability of our system to determine the appropriate level of the octree to use for processing volumes and rendering their triangles.

7.1.2 Hypothesis #1.1

"Such a system can provide and maintain high frame rates with immediate, uncluttered rendering of the finest resolution data of a model in the user's foveated direction of view with flexible run-time control over the extent of that data and of the rendering speed independently of the viewing direction, the size of the model in main memory, the use of specific graphics acceleration hardware, and whether the model is textured, derived from scanned data, an isosurface or from CAD, etc."

Our depth buffer strategy ensures that the fine resolution geometry which is rendered last has priority over anything that has been rendered beforehand, such as the navigation skin, wire-frame rendering of bounding boxes, or the wire-frame rendering of octree nodes. This was achieved by re-setting the z-buffer before rendering the fine resolution geometry. This strategy requires a single depth buffer and a single rendering pass.

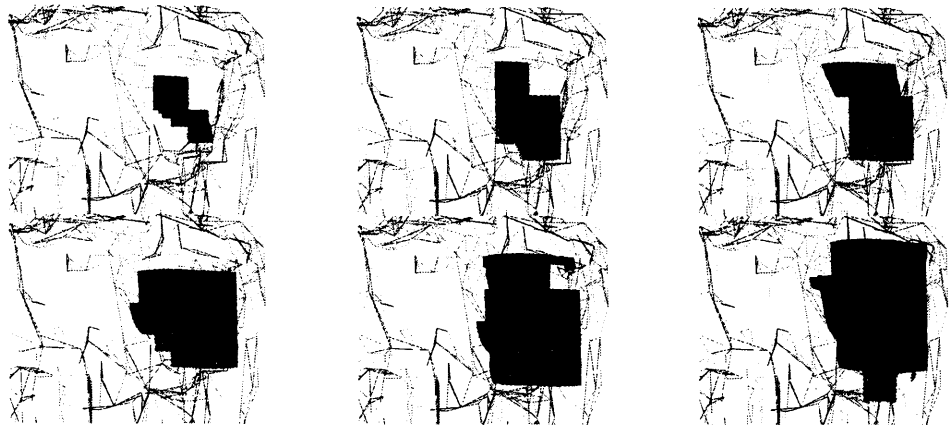


Figure 7.1: Increasing triangle rendering budget with a RenderArray of 3,458 nodes - *top left:* 4,000 triangles; *top middle:* 8,000 triangles; *top right:* 12,000 triangles; *bottom left:* 16,000 triangles; *bottom middle:* 24,000 triangles; *bottom right:* 32,000 triangles.

Figure 7.1 illustrates the type of flexible control the system grants the user at run-time. In particular, the user can temporarily increase the rendering budget at the expense of a decrease in frame-rate. When there is no interaction and no change of viewpoint, an idle function can gradually increase the rendering budget until the full model is rendered. Asychronous rendering allows the system to detect a change of the user's viewpoint, to pre-empt the current rendering process and restore the previous rendering budget so as to ensure continued interactivity.

Figure 7.2 in conjunction with Figure 7.1 shows how the user can trade a larger triangle rendering budget with a RenderArray of fewer nodes and lower spatial accuracy to obtain a rendering performance similar to that of a more accurate RenderArray with more nodes but fewer triangles. Specifically the extent of the area covered with a rendering budget of 24,000 triangles and a RenderArray of 3,458 nodes is larger than the area that would be covered with a more spatially accurate RenderArray of 32,502 nodes with a rendering budget of 4,000 triangles for a similar performance.

Figure 7.3 shows results of the Octree Interaction Engine (OIE) when inspecting the 56 million triangle model of David (1 mm scan) (b-d-f) and the 8 million triangle model of David (2mm scan) (a-c-e) with a 64-bit laptop with 2GB of main memory. An identical rendering budget of 100,000 triangles was used for a) and b). It can be seen from b) and d) that the same rendering budget maps to a significantly smaller area. The full resolution models are shown rendered overlayed on the navigation skin in e) David 2mm) and f) David 1mm with their respective clustered navigation skins of approximately 5,700 triangles in g) and h) respectively.

Table 7.1 shows some preliminary statistics that illustrate how developments in processing

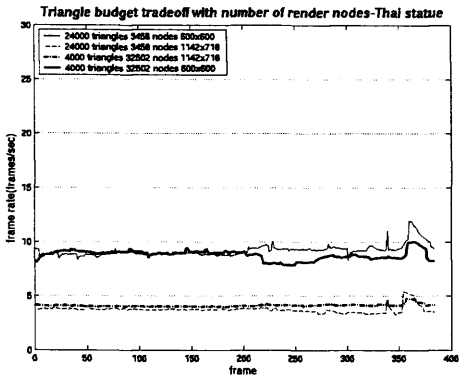


Figure 7.2: A 24,000 triangle rendering budget with a RenderArray of 3,458 nodes has a similar rendering performance of that of rendering 4,000 triangles with a more spatially accurate RenderArray of 32,502 nodes.

speed, graphics card capabilities and the advent of 64-bit architecture have enabled the Octree Interaction Engine to inspect increasingly detailed laser scanned models. We note that with the use of a personal 64-bit laptop computer it was possible to load and inspect the 56 million triangle model in 4 min 40s (7th row, middle column), enabling browsing of different models in rapid succession possible. The average frame rates were computed with similiar camera paths used in Chapter 3 of this thesis.

Figure 7.4 shows the results of the clustered navigation skin obtained with a variety of other scanned models, in addition to that shown in Figure 7.1.

In Chapter 3 we designed camera trajectories encompassing in our experiments the viewing of models from afar and from close-by. We evaluated rendering performances of models comprised numbers of triangles differing by several orders of magnitude and showed that it was possible to achieve a frame rate above 14 frames per second from any viewing direction with any model. We extended our rendering system to cope with textured models. The triangles from our triangle budget could easily belong to various large textures, the video memory required for the textures could be exceeded and the number of texture binds for texture rendering prohibitive. We created new smaller textures comprised of the fragments of textures used by triangles contained in octnode volumes. In addition at run-time a further sorting step on texture ids was added to our render loop, so as to avoid binding textures more than once.

Finally our compression algorithm PNORMS (see Section 3.5) and our compact data structures allowed us to read larger models to reside entirely in main memory, ensuring immediate access to fine geometry unlike the lags experienced with systems that rely on secondary memory

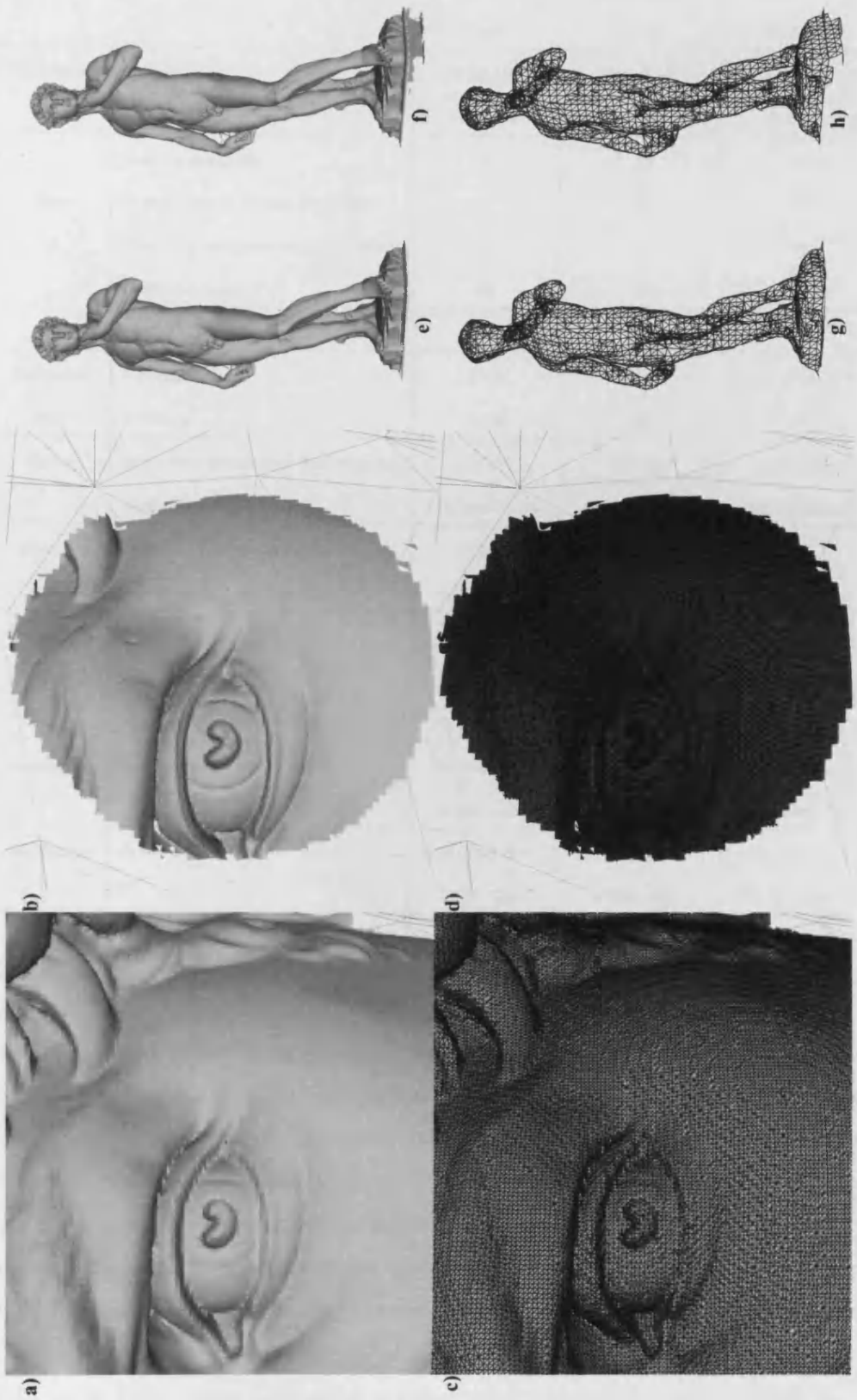


Figure 7.3: OIE: Inspection of the 56 (b-d-f-h) & 8 (a-c-e-g) million triangle model of David.

		David (2mm) *Intel	**David** (1mm) *Intel	**David** (2mm) *PowerPC
3D Model	#triangles	8,254,150	56,230,343	8,254,150
	#vertices	4,129,614	28,184,522	4,129,614
Time (s)	Read 3D model file	9	92	48
	Compression of normals (PNORMS)	11	76	77
	Clustering (navigation skin)+PNORMS	5	38	33
	Octree construction	10	78	111
	Total:	35	284	269
Navigation skin	# triangles	5,706	5,661	6,648
	# vertices	2,609	2,622	3,047
Octree	depth level reached/max #leaf triangles	12 / 100	13 / 100	10 / 100
	#leaf nodes / #total nodes	168,769/213,490	1,937,125/2,408,289	209,768/264,212
Memory (MB)	octree memory(total/node null pointers)	12.8/5.9	144.4/67	15.8/7.3
	total memory (model+octree)	230	1510	221
Rendering	# render nodes	31,627	13,597	2,832
	# triangle budget	100,000	100,000	4,000
	# average frames per second	31.1	37.8	20
Hardware specification	Laptop model name / RAM	Dell Precision M4300 / 2GB	Dell Precision M4300 / 2GB	Apple PowerBook G4 Titanium/ 1GB
	Processor	Intel Core2 Duo - T7300 2Ghz	Intel Core2 Duo - T7300 2Ghz	G4 - 500Mhz
	Graphics card (Rendering window size)	Nvidia (900x900) Quadro FX 360M	Nvidia (900x900) Quadro FX 360M	ATI (600x600) Rage Mobility 128

Table 7.1: Octree Interaction Engine (OIE) scalability on model size and hardware resources. It can be seen from the #render nodes row and the David(1mm)*Intel column that the OIE can adjust to sort a smaller number of render nodes to maintain the high frame rates than were possible with the smaller model David(2mm)*Intel. The benefits of using higher processing speeds and more powerful graphics acceleration can also be seen by comparing the right most column with the second and third column from the right. A larger triangle rendering budget of 100K triangles is possible when using a modern laptop versus a 4K triangle budget on an older machine with a larger rendering window of 900x900 pixels versus 600x600 and more render nodes sorted 31,627 versus 2,832 whilst obtaining high frame rates 31.1 and 37.8 versus 20.

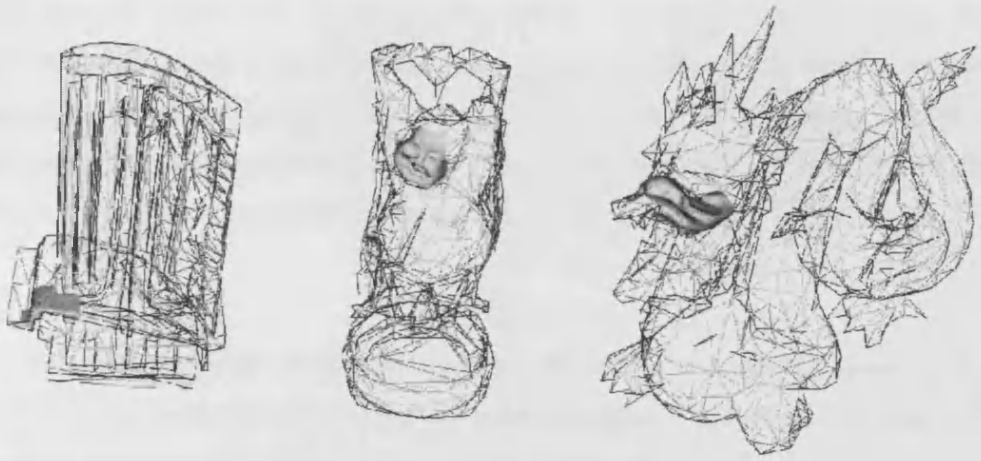


Figure 7.4: Other example of navigation skins.

to view the same geometry.

7.1.3 Hypothesis #1.2

"A system can view such models without any manual pre-processing and with less memory than required for the original models without a perceptible difference."

Use of our "Platonic Derived Normals for Error Bound Compression" algorithm allowed the 10 million triangle model shown in the middle of Figure 3.39 to be rendered with a maximum of 27,300 normals as opposed to 10 million that would otherwise have been required. The image in the middle of the figure has normals with an imperceptible maximum error of 2.5 degrees when compared to the original 10 million normals being used in the rendering of the left image. Our compression system also allowed one to choose between storing 2 bytes or 4 bytes per normal rather than the original 12 byte normals, enabling memory savings of up to over 83% which allowed us to test even larger models.

The textures computed for the volumes of the Octree Interaction Engine were also generated automatically.

7.1.4 Hypothesis #2 LoD cost functions

"In the context of non-uniform reduction, there is a vertex classification system in which the same heuristics and constraints can be used with success with various popular uniform reduction methods and choices of error function. Such a system can work with both regular, large scanned meshes and with irregular, small CAD meshes and preserve borders and colour discontinuities"

Our vertex classification system for non-uniform reduction (Chapter 5) was tested with

three different simplification algorithms using different semi-regular scanned models. Figure 7.5 and Figure 7.6 compare the geometric error of the memory-less quadrics approach applied to simplification of the Ciara body scan and Face scan respectively under uniform reduction and non-uniform reduction using our vertex classification heuristics. It can be seen that results obtained for the respective geometric errors are finely balanced.

We tested our vertex classification system in the context of completely preserving border edges of a model and of preserving colour discontinuities with a small CAD model. The required edges and discontinuities were successfully preserved.

Finally our system enabled us to preserve more vertices around the joint areas of the knees and elbows in a body scan, improving the rendering quality of their animation, for example as discussed in Appendix A by reducing the rendering artefacts.

7.1.5 Contributions

We presented solutions for pre-processing 3D models in order to remove obstacles to simplification of their geometry. In particular, we have presented solutions for cleaning duplicate vertices in 3D models that hinder reduction of their geometry as well as an algorithm to consistently orient the normals of the constituent triangles, the latter/both even if the models contain non-manifold triangles. Both are frequent problems: the former in poorly exported CAD models constructed from a number of sub-parts, and the latter in models obtained from scanned real objects as surface reconstruction algorithms often give a higher priority to finding a surface that approximates a point cloud than to finding the orientation of the resulting surfaces.

Within the context of quadric-error based LoD algorithms, we presented an automatic condition number calibration method to control location of the optimal placement of the new vertex after an edge collapse. This was a challenging problem as many other approaches did not succeed in detecting an appropriate condition tolerance that would work over a variety of models, of different types, scanned or CAD, independent of model scale, and for a variety of different kinds of quadric error and LoD approaches. We hope that people working in other areas using numerical solvers such as singular value decomposition will appreciate from our work how a poorly chosen tolerance can inadvertently create sub-optimal results. We hope also that, as a result of our work others will realize how the choice of units in which a model is represented can affect how a problem is diagnosed. In particular, in our work we saw that often a problem might erroneously be diagnosed as ill-conditioned when in fact an optimal solution can be easily found. In addition, suppliers of 3D modelling tool packages could implement our

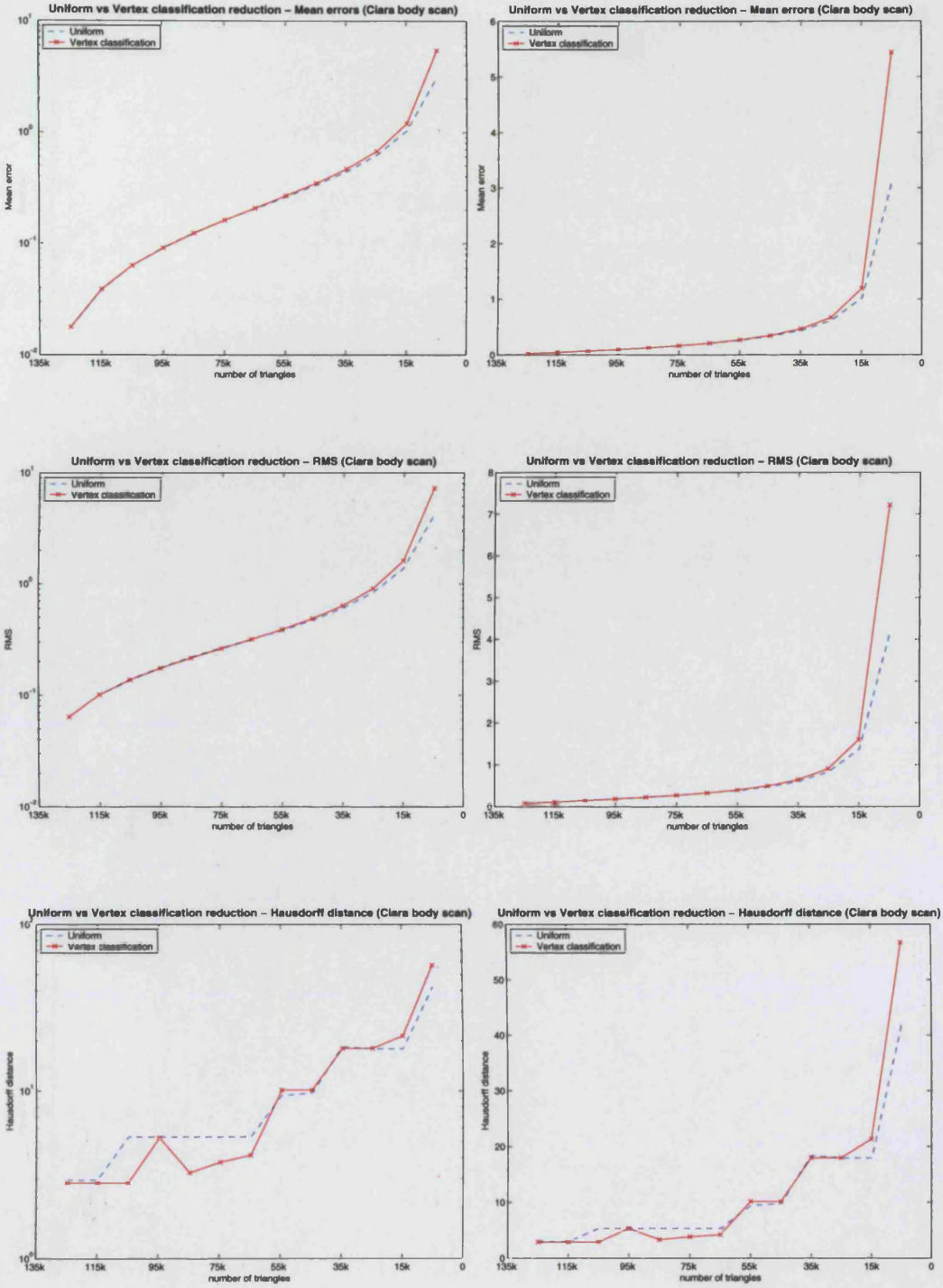


Figure 7.5: Uniform reduction versus non-uniform reduction for the Ciara body scan model - the results of both methods of reduction are finely balanced, left, log scale plot of errors and right, linear scale plot of errors.

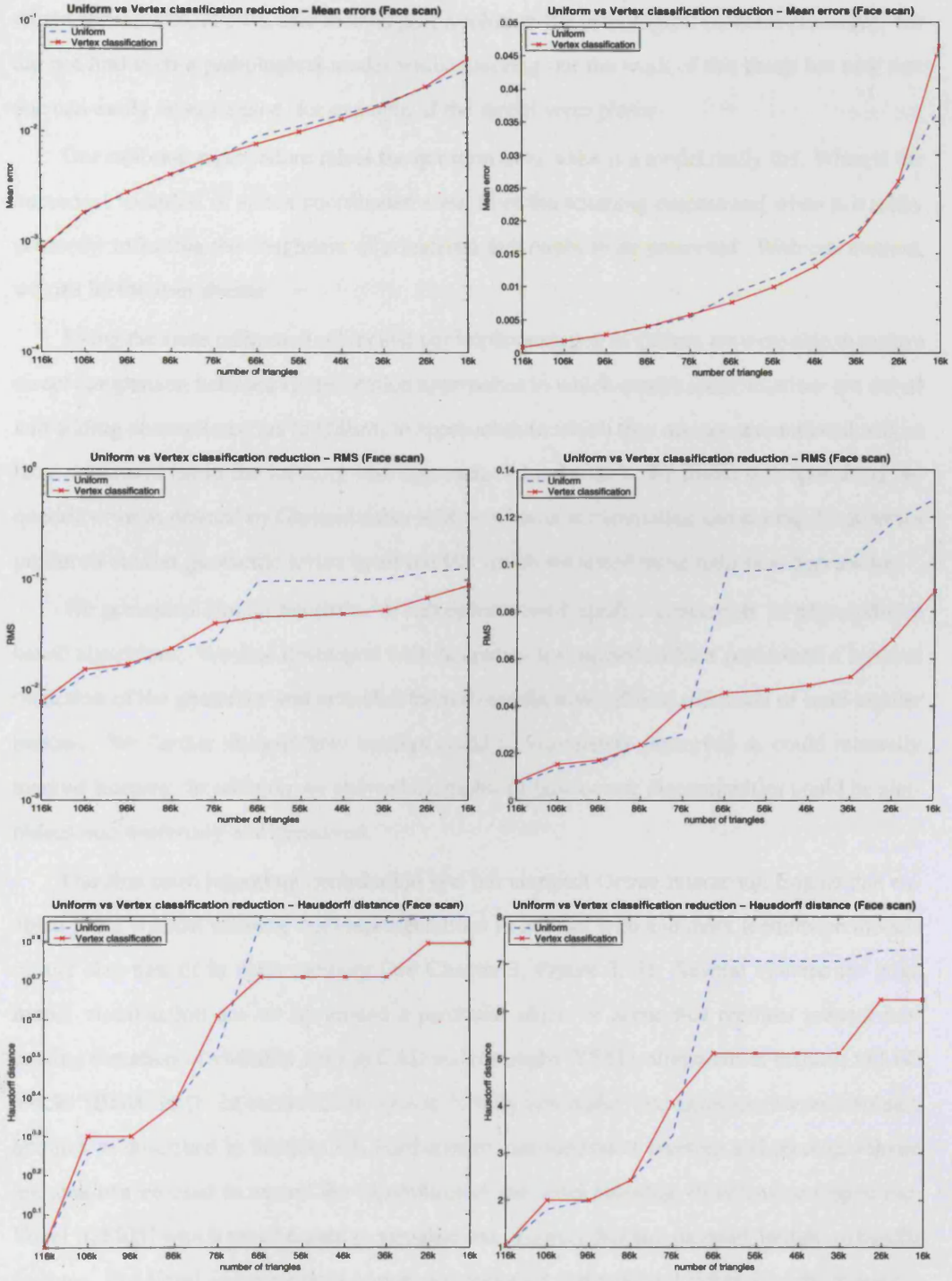


Figure 7.6: Uniform reduction versus non-uniform reduction for the face scan model - the results of both methods of reduction are finely balanced, left, log scale plot of errors and right, linear scale plot of errors.

calibration procedure in order to provide an initial, or 'default' solution to such problems and, where appropriate, allow a user to hand pick a solution if a pathological model is presented. We did not find such a pathological model whilst carrying out the work of this thesis but note that one can easily be envisaged, for example, if the model were planar.

Our calibration procedure raises the question as to when is a model really flat. When is the numerical variation of vertex coordinates noise from the scanning process and when is it really geometry reflecting the roughness of a material that ought to be preserved. With our method, we can let the user decide.

Using the same calibrated solver and our implementation of QSlim, we were able to make a direct comparison between simplification approaches in which quadric error matrices are stored and adding accumulated (as in QSlim) to approaches in which they are not accumulated and, in fact, not stored (as in the memory-less approach of Lindstrom). We found that optimising the quadric error as defined by Garland either with or without accumulating and storing the quadrics produced similar geometric errors in all models which we tested these reduction approaches.

We presented simple heuristics to incorporate mesh quality constraints in edge-collapse based algorithms. We first developed such heuristics for methods which performed a uniform reduction of the geometry and extended them to handle non-uniform reduction of semi-regular meshes. We further showed how borders could be completely preserved as could manually marked features. In addition we showed examples of how colour discontinuities could be simplified non-uniformly and preserved.

Our first most important contribution was our compact Octree Interaction Engine that enabled users without creating LoD representations to interact with and mark features on models of any size that fit in main memory (see Chapter 3, Figure 3.11). Several systems for large model visualisation are set up around a particular object or scene that requires manual processing (creation of visibility cells in CAD walkthroughs [TS91], alignment of textures to LoD blocks [BGB+05]). In contrast, our system is fully automatic, and generates its own volume textures as described in Section 3.3. Furthermore, our method of creating and binding volume textures can be used to extend the capabilities of the voxel splatting visualization engine Far-Voxel [GM05] which may be used to visualise out of core CAD and scanned models, to handle textures. Far-Voxel *splats* vertices (points are projected and rendered rather than the triangles) when the projected size of a voxel is small and renders geometry when the projected size of a voxel is larger than a user set tolerance. Specifically, in order to extend Gobbetti et. al's [GM05] system to visualize out of core texture models, one could automatically break large surface textures into new smaller and more manageable volume textures (as in Section 3.3). One could

then sort on texture id the smaller voxels that index the volume textures higher up before rendering. This last sort could also be performed in Far Voxel's precomputation phase, just before the creation of triangle strips.

We believe that there are a number of visualization applications and diverse platforms that would benefit from not using secondary memory to view models. For example, in applications based on the use PDAs and mobile phones with large capacity flash memory. In addition museum curators responsible for making and keeping a 3D digital record of their artefacts could benefit from quick inspections using our tool to determine whether further scanning was necessary in any area.

Our second major contribution was our normal and colour compression algorithm PNORMS: Platonic derived normals for error bound compression. This was described in Chapter 3/Section 3.5 where we showed that a five times subdivided icosahedron produced two and a half times as many normals as obtained from the subdivided octahedron often used in the compression literature. We showed and made available our databases of normals that had better distributions than other solids. In addition we provided code that showed how to use our algorithm in a rendering system with 83% memory savings on the normal attributes in comparison to systems that used uncompressed normals calculated anew for each triangle. One aspect with which we are particularly happy with is the fact that our approach uses byte aligned normal ids which thus do not require decompression thereby enabling the cpu or gpu to carry out other tasks with no performance hit. We envisage that several applications can benefit from such compression with imperceptible differences when using normals with a maximum encoding error of 2.5 deg or 1.3 deg according to whether fast encoding or slower but more accurate encoding was used respectively. For applications requiring greater precision, databases of normals with encoding errors of less than a degree were also made available.

Two major and several minor contributions have thus been made as described above, but these do not exhaust the possibilities that the approaches we have adopted suggest. We thus present several areas of potential future work in Section 7.3.

7.2 Summary

This section summarizes the thesis as a whole (Figure 1.4).

Features for non-uniform geometry reduction can be either detected automatically without visualization (such as the border edges of the Standford bunny on the left of the image), or can be defined manually.

Non-uniform reduction techniques assume that one can easily interact with any sized

model and manually mark features for their preservation. This is clearly not the case for many models. Rendering acceleration techniques that use simpler representations (LoD) create uniformly reduced meshes, however one wishes to select fine resolution triangles. This situation creates a chicken and egg problem where one needs LoD to accelerate the rendering, but needs to have access to the fine unreduced geometry to preserve it. Furthermore, the triangles displayed in an LoD system are not guaranteed to be of the finest resolution, but one wishes to preserve the finest version. This problem is exacerbated when a viewer makes quick changes of view direction when inspecting a model residing in secondary memory, as content delays prohibit the immediate selection of the fine resolution versions.

We present a visualization system which we termed the Octree Interaction Engine (right of image) which addresses the interaction and selection problems mentioned above. A run-time user-adjustable triangle rendering budget of fine resolution geometry is always guaranteed to be rendered at the users foveated direction, and a depth buffer strategy ensures that these fine resolution triangles are always rendered unobstructed from any LoD used. To ensure immediate access to fine resolution geometry we create a single clusterd LoD mesh, which we term the navigation skin to provide an overview of the extent and shape of a model. We developed normal and colour compression algortihms to enable the visualization of larger models. We note with the advent of 64 bits personal computer machines, the models that once would only fit in out-of-core memory are now increasingly fitting in main memory. Our visualisation system that does not have the memory overheads of traditional LoD systems can visualize any model that fits in main memory at approximately the same speed from any view direction. Our system can view textured models without any manual pre-processing unlike existing visualization systems. Our visualization system can load and visualize rapidly unprocessed models unlike existing triangle based LoD systems [BGB+05, CGG+04] thus our system supports fast browsing of multiple large objects.

Before we could develop a general non-uniform reduction framework that would work with several uniform simplification algorithms, we addressed and solved numerical issues that occur with the numerical solver of quadric based uniform reduction systems.

In order to simplify a broader range of models, algorithms for geometry cleaning were developed. Namely for the deletion of duplicate vertices that generates cracks in a mesh, and for consistently orienting the normals of a model.

Finally we showed how our vertex classification system could be used to preserve features of semi-regular scanned models. For example for retaining more vertices around knee and elbow joint areas that stretch under soft deformation. An animation system which computes

bones automatically for body scans was developed to evaluate the benefits of our non-uniform reduction results.

7.3 Future work

We plan to extend our Octree Interaction Engine to a multithreaded asychronous system, in which a number of rays are cast into a scene and each ray would be processed by a different thread. Rays would be cast close to the user and afar. Figure 7.7 illustrates the potential for exploring different ray prioritisation strategies in the context of the navigation of massive models.

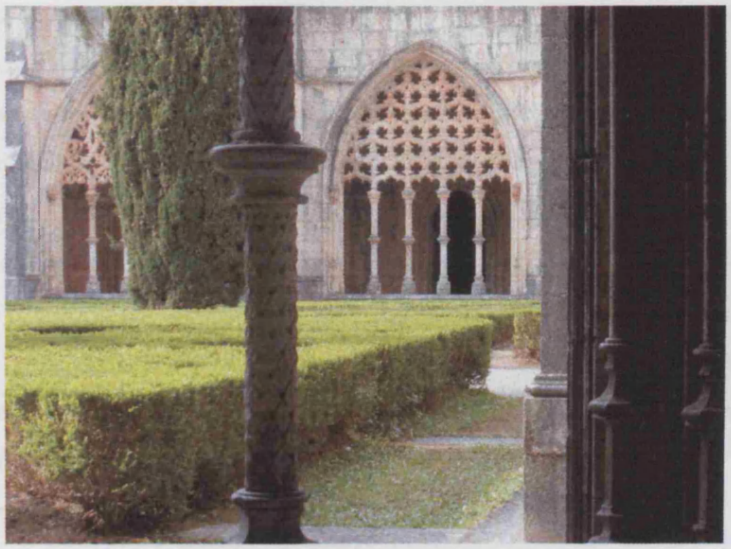


Figure 7.7: Mosteiro da Batalha - Photograph illustrating detail close to the viewer and detail far away which is important for navigation. If in a graphics model of this scene we were to attribute part of the triangle budget to fine resolution triangles in the area near the intersection point of the line of sight and the door area it could act as navigaton cues as in the real world.

In addition we would like to explore parallel rendering using a cluster of machines in which each machine would have a copy of the massive model an indentical copy of the RenderArray, casted its own rays processed by their own threads and rendered into their respective viewports so as to form a combined, larger image that could be projected on a large screen, or 'wall' by means of a multi-projector system. We could explore increasing the number of navigation skins, incorporating LoD techniques as in [BGB+05].

We would like to further improve the coherence/front span [IL05] of our models for mesh streaming by in addition to our triangle re-ordering re-label the vertices within each octnode, for example as in Cignoni et al. [CRMS03].

Even though our Octree Interaction Engine performed well with both textured and untextured, large scanned models, we would also like to experiment with other visualisation techniques for CAD models whilst keeping the memory footprint small. We have shown that superimposing wireframe rendering of the different levels of the octree can give the user a useful idea as to where different components of a model lie. We would further like to experiment with alpha blending the solid-rendering of octree nodes of different levels. Each octree node would be rendered with a colour approximating the colour of the geometry enclosed.

In addition we have shown that an octree that is built aligned to a CAD object's principal axis, can create render-nodes that when rendered provide a useful approximation of the model, which we call a *navigation volume.* This navigation volume like the surface *navigation skin* provides an overview of the shape and extent of a model, whilst using very little memory.

We would also like to experiment with using a hierarchical regular volume grid [GM05] which would enable even sized render-nodes, whilst retaining our octree construct. The reason for exploring this avenue, is that scanned models with their almost uniform distribution of surface points typically yield evenly sized render-nodes that align well with a model. In distinction, CAD models comprised of multiple objects of different sizes can have quite sparse geometry in comparison to scanned models, consequently CAD models can generate very different size render-nodes which do not approximate well the model when the render-nodes are rendered in full. Even sized render-nodes could be used hiearchically to counter the sparsity of the underlying model.

Appendix A

Application: Non-uniform geometry in an Animation system

This chapter presents an example of non-uniform geometry reduction in the context of an animation system. The animation system was published in [OZSB03] and was co-developed with colleagues Dongliang Zhang and Bernhard Spanlang, and is presented here for completeness.

The work presented here shows how non-uniform geometry reduction can improve the rendering quality of deformation areas in animation by retaining more detail in those areas. The automatically detected feature areas around knee and elbow joints used in the body scan model example in Chapter 5 were obtained by using a simple distance tolerance to the planes formed by skeletal joints and surface landmarks. This appendix shows how the geometric form of such joints and bones can be computed automatically.

We present techniques for automatically creating and animating models obtained from human whole body scanned data. A layered model is developed in which the underlying skeleton, simplified surface and mapping of the surface to the skeletal structure are generated without manual intervention. External body landmarks such as those employed in anthropometric surveys are used to obtain the skeleton, these landmarks are then reused to help map the model surface consistently to the skeleton. The main contribution of this work is to show how automatically or even manually extracted surface landmarks greatly simplify the task of skeleton generation and surface bone mapping, the latter uses surface connectivity to grow surface regions as opposed to conventional volume searches. The techniques are illustrated by animation from motion capture data in which a vertex blending technique is used to create continuous and smooth deformation.

A.1 Introduction

Real-time computer generation and animation of realistic 3D human models presents several research challenges. For example, in an application where a small number of high fidelity models are to be used in close-up, whether it be in entertainment, communications, or a virtual environment, it is necessary that the model's appearances are personalised, in shape and look, to resemble real humans and that they move and deform smoothly and realistically. This is especially the case if the models are to be used to represent specific individuals, but it is also true whenever a number of distinct characters are required in order to populate a computerised world, and remains true to some extent even when many electronic characters are required in crowded scenes if undesirable artefacts are to be avoided.

However, creating and animating such models is often a time consuming task, requiring detailed input from skilled modellers, animators and creative artists. Although a number of systems have been developed for such applications in computer graphics, there are none that can deal with the high resolution 3D scans of real people that are becoming increasingly available. For example, human whole body scanners have been produced by Cyberware, TC2, Virtronics, Wicks and Wilson, Hamamatsu and others, often for anthropometric purposes or for applications in the fashion and clothing industries [DDBT99]. Such systems may be used to produce scans of particular individuals or of as many people as needed, for example for sizing surveys, but the data they provide is usually a static image of the body skin or of some close fitting undergarment.

The aim of this work is thus to bring together and develop techniques that would enable such high resolution scans to be used automatically to produce the layered human models frequently used in animation. This requires that the scans be segmented, that joints be identified and a skeleton-like structure inserted inside a scan, that surface points be mapped to each section or "bone" of the skeleton, and that the surface be deformed appropriately as the skeleton is moved to animate the model. In addition, it is frequently necessary to simplify the models created from the scanned data in order to reduce the computational burden in many animation applications, especially those using a number of models and/or needing real-time operation as in a virtual environment. Level of detail representations of such models therefore also need to be produced, in particular [OB01], to reduce the number of triangles from over 100,000 in the original model, to a few hundred triangles without loss of joint articulations or of limbs. In this study, the human model is represented by a triangular mesh and described as indicated above, by a layered model [SHSI01]. The model is constructed from a point cloud, which is obtained from a Hamamatsu Body Line Scanner [Hor95], and refined by using the techniques

of surface reconstruction [DKW+98] and mesh simplification [OB99, OB01]. We use fully automatic techniques for segmenting the 3D body scans and locating their key landmarks as published [FGP98] and [BDDV02]. These surface landmarks are close to the joint locations that we wish to articulate as shown in Figure A.6. Our main contribution from this study is to show how automatically or even manually extracted surface landmarks greatly simplify the task of skeleton generation (Section A.3.3) and surface bone mapping (Section A.3.4). Finally we apply a vertex blending technique to generate smooth human deformation. We have effectively applied our method to a variety of human models, ranging from male and female adults to models of children (Figure A.16). This would not be possible if the techniques required manual intervention at any stage. The structure of this appendix is as follows. In the next section, we briefly review previous work on human animation. In Section A.3, we introduce our layered human model, review the feature extraction and segmentation in Section A.3.1, mesh reduction in Section A.3.2, in Section A.3.3 we present the automatic skeleton generation method, in Section A.3.4 we present our novel surface bone mapping technique and in Section A.3.5 we describe the motion capture system. In Section A.4, a vertex blending technique is employed to deform the human model, and we present how to compute a weight function for the vertex blending. Finally, in Section A.5 we show some results, including the use of non-uniformly reduced meshes. We summarise our method and discuss future work in Section A.6.

A.2 Related work

Animation of realistic 3D human models typically involves three main tasks: a) modelling of the character: skeleton creation, skin and muscle representation, possibly cloth; b) deformation: rigid deformation of the skeleton, soft deformation of the skin and possibly muscles; c) motion control: motion capture, key-frame interpolation, behaviour interpolation, control point guidance, and animation parameters or scripts. Several human animation systems have followed a layered approach to represent skeleton, muscle, skin and control [CHP89, KMTM+98]. For deformation of muscles, and skin, techniques derived from free form deformation (FFD) of spline control points have been used. Thalmann et al. [ST95, TS96] try to personalise avatars with a few thousand polygons by using metaballs and scaling the model via five or six parameters. For higher fidelity it would seem natural to use scanned body models. Nebel [Neb00] constructs volumetric meshes of soft tissue above the muscle from photo realistic surface scans. The surface skin mesh is replicated three times to form epidermis, dermis and subcutaneous tissue. A finite element method (FEM) is used to model the different physical properties of the layers, and external forces are then applied to simulate skin deformation. Chen et al. [CZ92] use a

FEM to model muscle deformation, Scheepers et al. [SPCM97] go one step further to model anatomy based muscles that deform the skin. Mapping the skin vertices to underlying bones or muscles often relies on a skilled animator [MMT97]. The task of surface to bone mapping can also be quite cumbersome for animators due to the global nature of querying planes. We adopt a surface growing technique that does not inadvertently select other unrelated parts of the model (Section A.3.4). Teichmann [TT98] has developed a semi-interactive process to define the skeleton of an arbitrary object, based on a closed mesh. This method produces good articulated models, but requires filtering of medial axis results and still some manual intervention to help guide the skeleton creation process. Similarly the result obtained when using the medial axis [ABK98] output option of the freely available software Cocone [Dey02, DG03] also is not immediately usable as a skeletal structure, as can be seen in the rightmost image of Figure A.1. Although the produced surface represents outstandingly well the underlying point cloud, the medial axis is comprised of both line segments and triangles making it difficult to establish unique positions for joints.

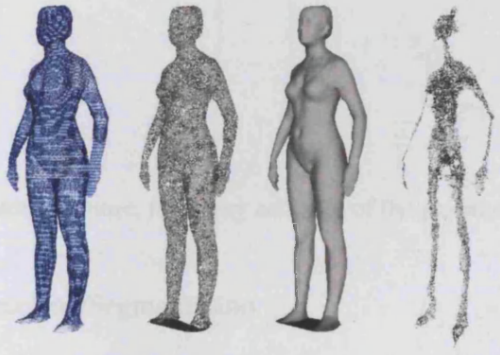


Figure A.1: Surface reconstruction and medial axis - from left to right: point cloud; surface produced by Cocone; Cocone surface with corrected normals using our algorithm for fixing inconsistent normals of chapter 6 section 6.1, noisy medial axis produced by Cocone .

In Section A.3.3 we show how automatic surface landmarks can find useful approximations of the medial axis for skeletons. Sloan et al. [SRC00] combine shape blending with simple transform blending for soft deformation of the animated surface. In their system, muscles can contract via target shape position interpolation and at the same time move with the bones. Motion capture data can then be used to drive the skeleton animation. For example Fua et al. [FGP98] detect and derive animation parameters from analysing motion video sequences. These motion capture sequences can then be interpolated with other sequences and altered to have expression such as happy or sad [RCB98].

A.3 Layered Human Model

In our work, the human model is considered as a deformable, articulated layered model. We use two layers: skeleton and skin. For the purpose of applying motion capture data to our model, the skeleton resembles the hierarchy of the source motion capture sensors. The structure and hierarchy of the skeleton that we wish to build is shown in Figure A.2. Subjects being modelled are typically scanned with a stance that is suitable for the initial pose matching of motion data such as the BioVision hierarchy (BVH) format [bio03]. Each skeleton segment is described by a local reference system used to define its local position and orientation as also discussed in Section A.3.4. The following subsections, describe how we build the matching layered model.

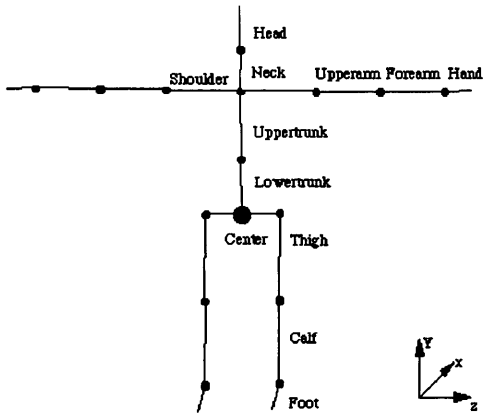


Figure A.2: Skeleton structure, hierarchy and axes of the global co-ordinate system

A.3.1 Feature Extraction/Segmentation

In all the examples used here, the whole body data was obtained from a Hamamatsu scanner Horiguchi [Hor95]. Data from this scanner is conveniently represented as a set of horizontal data slices made of 3D vertices [BDDV02, Hor95]. If data from some other system such as that developed by TC2, say, is to be used which is not represented in this way, one can use octrees to query and convert the object into equivalent slices. Once the data is in this 'slice' form, and provided it is of sufficiently high resolution to delineate the requisite details of the anatomy, the landmark detection and segmentation process discussed in [DDBT99, BDDV02] may be used, we will only briefly review the process here. At a high level, primary landmarks are detected such as the top of the head, torso, neck, left and right armpits, crotch, and the ends of the arms and legs. These landmarks can then be used to create smaller search volumes for secondary landmarks such as the wrist (Figure A.5 right). The first landmarks are detected by algorithms such as that designed to locate the armpits from a re-entrant surface condition as illustrated

in Figure A.3 (left). The centroid of each horizontal data slice is calculated, and the vertices binned into sectors of angular width β, which is related to the number of samples in the slice. The arms can be segmented from the torso by detecting the transition slice that indicates the branching point at the armpits. Sector bins will typically contain only a few vertices, but when the branching at the armpit occurs, a tolerance can be set to detect the increase in number of vertices in a bin. Similarly the neck and crotch can be detected by reasoning about the average distance to the centroid from slice to slice, and about changes in depth [DDBT99].

The detected primary landmarks define sub-search areas on the surface of the body scan in each of which a variety of discriminant functions use local shape characteristics like local maximum curvature to detect other landmarks such as: shoulder, elbow, wrist, waist, hip, knee and ankle. This algorithm is part of the system developed to automatically process the Hamamatsu body scans for applications in medicine and in the retail and clothing industries [BDDV02]. Another approach for detecting for example the underarm point for segmentation could be to project contours, given that the stance of the bodyscan is controlled, and detect local maximas of the contour as illustrated in Figure A.3 (right).

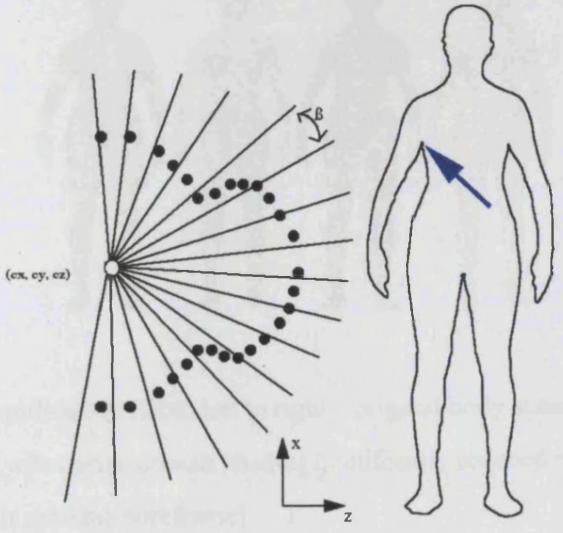


Figure A.3: Underarm segmentation - *left:* using re-entrant point tolerance criteria *right:* projecting contours and detecting local maxima (blue arrow).

Alternatively one can simply select manually vertices in the surface of the model that are close enough to the automatically detected landmarks mentioned above.

A.3.2 Mesh Simplification

Applying deformation matrices to all the 67,595 vertices of the original model is computationally unfeasible for interactive animation rates and in addition, rendering the associated 135,099 polygons would also prove to be a problem. To overcome these difficulties, we used both the uniformly reduced models and non-uniformly reduced models. The feature detection step of the simplification method of Oliveira and Buxton [OB01] yields a robust and clean approximation to the medial axes of the body parts, which can later be used for finding the appropriate joint positions of the skeleton. The medial axes approximation consists of a set of computed centroid vertices from each slice of the high resolution scan (Figure A.6). The following figure, Figure A.4, shows the original body scan surface on the left, and on the right, the uniformly decimated surface reduced to ~3.7% of the data quantity (4,999 polygons), which is used in our animation. We should be able to articulate and animate crowd scenes that require high quality, low-level of detail rendering of humans (190-250 polygons/ avatar), for example, by texture mapping facial details and clothing on to our models as in the work of Hilton et al. [HBG+99].

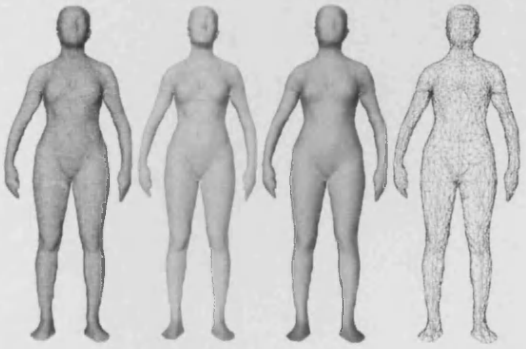


Figure A.4: Mesh simplification (from left to right): original body scan model of 135,099 triangles; 67,595 vertices [wireframe/smooth shading]; uniformly reduced model of 4,999 triangles; 2,535 vertices [smooth shading/wireframe].

A.3.3 Building the Skeleton

We can now use the detected surface landmarks of Section A.3.1 (Figure A.6, left) together with the axes which we regard as defining an approximate medial axis, from Section A.3.2 (Figure A.6, middle) to find the skeleton. We use only a subset of the detected landmarks, namely we only use the landmarks that are deemed to be close to the joints of the motion capture hiearchy (Figure A.2) that we wish to use. (see third image from left in Figure A.5).

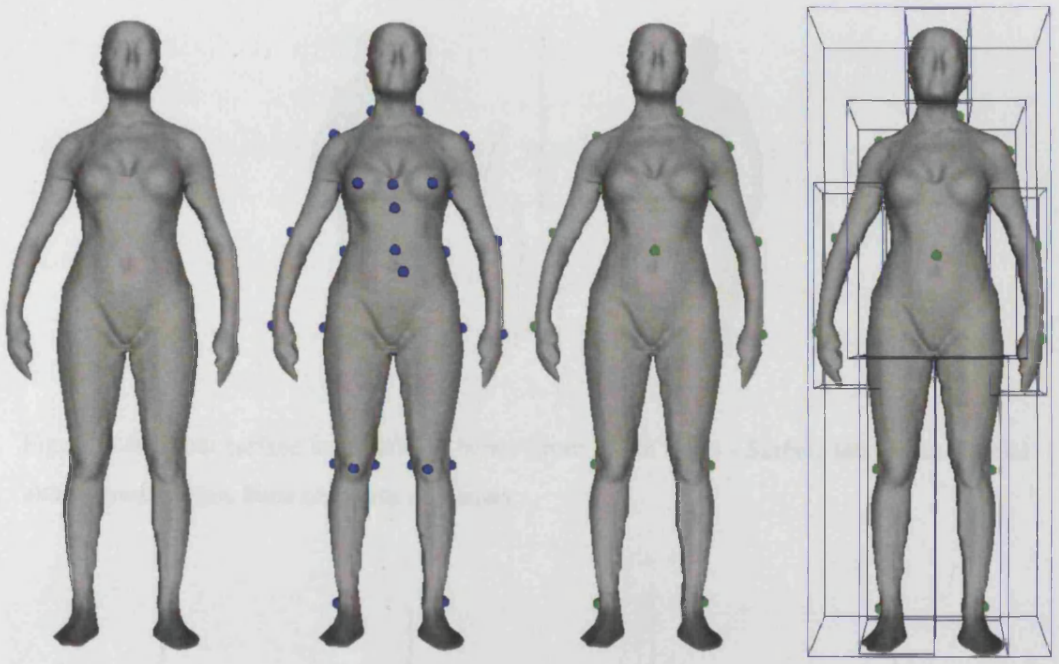


Figure A.5: Surface landmarks - from left to right: original model, primary and secondary landmarks, landmarks compatible to the motion capture skeleton, landmark search volumes.

For each surface landmark (neck, chest, waist, shoulder, elbow, wrist, knee and ankle etc.) we compute the distance to each vertex in the approximate medial axis, and we choose the vertex on the approximate medial axis with the shortest distance to be the corresponding joint position. The approximate medial axes described in Section A.3.2 was generated by joining the centroids of all the horizontal data slices of the model. Since the model is segmented as described in Section A.3.1, the search on the approximate medial axis set for the point that represents the joint corresponding to a surface landmark is reduced to under 200 distance calculations. To conform with the hierarchy of the motion capture data, some joints such as the uppertorso (uppertrunk) and abdominal (lowertrunk) joints (Figure A.2) are determined by making use of the ratios of skeleton segments [Ass93]. Figure A.6 (right), shows the skeleton generated automatically from the scanned human model, based on these key landmarks. The various stages of creating the bone structure are perhaps best illustrated from left to right in Figure A.7.

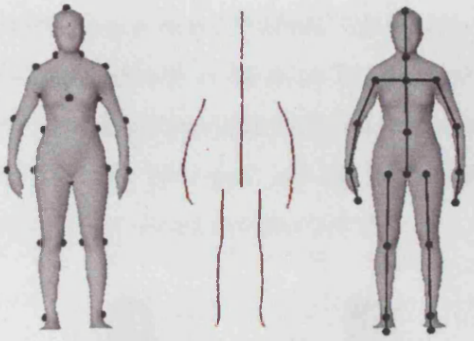


Figure A.6: From surface landmarks to bones (from left to right) - Surface landmarks, medial axis approximation, bone segments and joints.

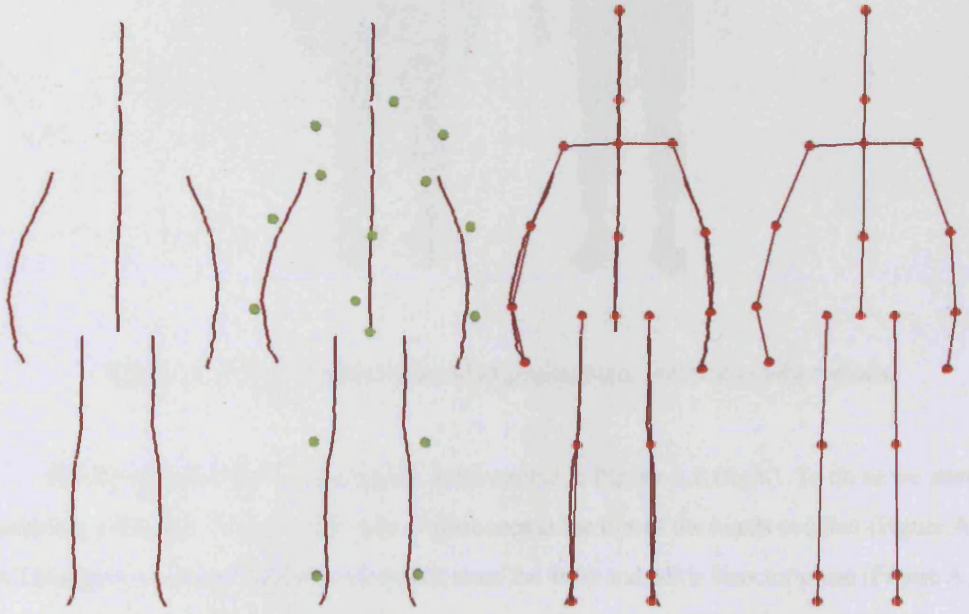


Figure A.7: Bone creation (from left to right) - Medial axis approximations/centroids of slices, medial axis approximation and used landmarks, medial axis approximation + closest points to landmarks of medial axis approximation + bones connecting the closest points, bones connecting the closest points.

A.3.4 Mapping

After we have built the skeleton layer, we divide the surface mesh layer into several parts based on the skeleton, and then, map the vertices of the surface into their corresponding skeleton segments. In this process, each surface vertex (of the decimated model) is labelled as belonging

to a particular segment referred to as a 'bone' [FGP98]. This is done by placing at most joints, bi-sector separating planes, perpendicular to the plane formed by bone pairs, and testing the vertex position in relation to the plane/bone segment by placing at the joints, bi-sector planes, perpendicular to the plane formed by bone pairs, and testing the vertex position in relation to the plane/bone segment [SHSI01] as shown in Figure A.8 (left).

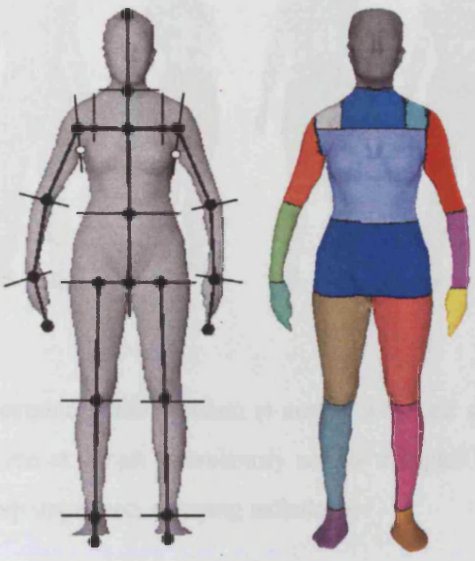


Figure A.8: *Left:* Surface separation planes, *right:* surface grown regions.

Finally we grow the surface region as illustrated in Figure A.8 (right). To do so we start by selecting a triangle closest to the ends of the bones at the tips of the hands and feet (Figure A.8). All triangles connected to these that do not cross the wrist and ankle bisector plane (Figure A.11-a) respectively belong to the hands and feet. A similar procedure is then applied to the lower arms and legs, etc upwards, until every vertex in the scan is assigned to a region. Two additional planes are created perpendicular to the shoulder bone (Figure A.8), approximately 3/5 of the distance from the chest joint to the shoulder joint. These are to accommodate complicated motion of the body in the shoulder region. In addition problems may arise when the orientation of a bisector plane would lead to it intersecting the interior of the body segment (Figure A.9 b) & c)). In particular, planes at the shoulder do not bisect the joint angle but are constrained to contain the armpit landmarks as shown in Figure A.8 (open circles) and in Figure A.11-b). To stop the surface from growing to the torso we introduce a two step mapping procedure for the upper arm. Specifically, triangles that are just below the armpit landmark in the upper arm are selected, then triangles above the set height and behind the perpendicular plane that passes

through the armpit landmark and shoulder joint are also selected (Figure A.10, right).

Other methods would typically require the animator to tweak the separating planes so as to not select other parts of the body, or require the user to create additional constraining planes, which can be difficult to control and visualise.

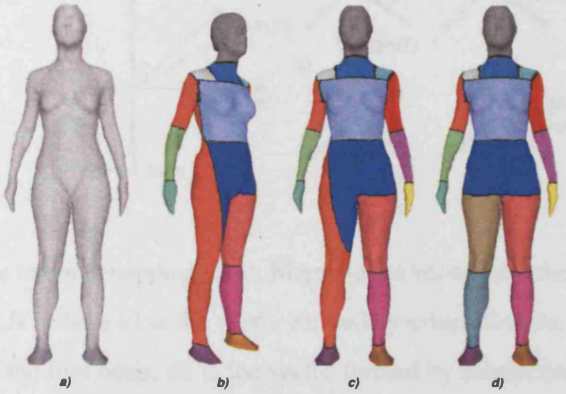


Figure A.9: Surface separation plane problem at armpit with one step mapping - *a)* original model; *b)* & *c)* global plane at armpit errorneously selects triangles from torso and leg too; *d)* mapping with our two step upper arm mapping technique.

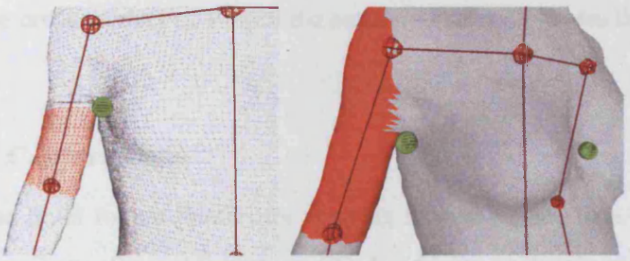


Figure A.10: Two step upper arm mapping - *left:* first step: triangles are added to the surface in brown vertically upwards from the elbow until a height just below the spherical armpit landmark in green, thus excluding triangles in the torso because they are not connected; *right:* second step: triangles that are above the armpit landmark and behind a perpendicular plane to the bones and passing through the landmark and joint are added to the brown surface.

The mapping between the surface and skeleton, sets joint and bone proximity parameters for each vertex that will be used in the blending deformation process described in Section A.4. Specifically the mapping consists of calculating three parameters for each vertex: u, r and q, where u is the ratio along the skeleton segment and its value is between 0 and 1, r is the distance from the vertex to the skeleton segment, and q is the angle which represents the orientation

related to the skeleton segment. From Figure A.11, we see that the mapping is based on a cylindrical coordinate system [SHSI01].

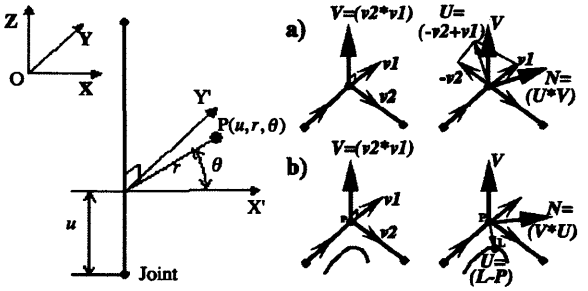


Figure A.11: Surface to bone mapping - *left:* Mapping the vertex onto the bone, *right:* the types of separating planes N, where $v1$ is the vector formed by subtracting the second endpoint from the first endpoint of the first bone, $v2$ is the vector formed by subtracting the second endpoint from the first endpoint of the second bone and V is an auxiliary vector used to build N which is defined by the cross product of $v2$ and $v1$: a) *joint bisector* where U is an auxiliary vector formed by adding the vector $-v2$ to $v1$, the cross product of U with V creates the bisector separating plane N, b) The landmark point L (*e.g.* armpit of the rightarm) is used to constrain the separation plane N, U is defined by subtracting the landmark L from the second endpoint of the first bone. The cross product of V with the auxiliary vector U creates the separation plane N.

A.3.5 Motion Capture Data

Finally we use the BVH format [bio03] for applying motion capture data available from the avatar modelling package, Poser 4.0 contents CD. In addition to the skeletal hierarchy of the bones and an initial pose, this format describes each motion frame as a sequence of x, y, z positions and the Euler angles of the skeleton joints. The sequence of parameters can then be applied to each bone segment's deformation matrix. We adjust the local matrices in the beginning to reflect the initial pose of the motion capture data, and then refresh the angle deformation to produce the different animation frames. We apply angular interpolation if the motion capture data available is not smooth enough.

A.4 Vertex Blending

In order to ensure that the body surface does not break up as the skeleton moves, the above process requires that the surface is deformed and blended at each vertex as illustrated in Figure A.12.

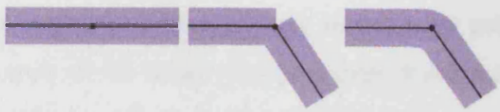


Figure A.12: (a) Original shape (b) Rigid transformation (c) Deformation using vertex blending.

In our method, we use a vertex blending technique [Die99] to create continuous and smooth human deformation. To make this possible, several transformation matrices are used, each matrix exerting influence on its part of the object. For example, the first matrix can twist the left part of a limb in one direction while the second matrix does the same to the right part in the opposite direction. Apart from the transformation matrices themselves, one must set weight factors for each vertex. These weight factors determine the influence exerted on the vertices of the skin by each matrix and are set as vertex attributes along with normal, colour, texture coordinates, etc.

By way of an example let us consider the idealised model shown in Figure A.12-c. The model consists of two parts. Even though the two parts move independently, the model remains continuous and smooth. The model is transformed by two matrices - one for each model part. The matrices are based on their bones. Vertices belonging only to one bone are simply transformed along with the bone. Vertices belonging to two or more bones are transformed by several matrices after which the result obtained is averaged in proportion to the weight factors. The mathematical expression of the vertex blending for two transformation matrices may be described as:

$$\begin{aligned} x = &\quad T_0 x_0 w + T_1 x_0 (1-w) \\ N = &\quad T_0 N_0 w + T_1 x_0 (1-w) \end{aligned} \tag{A.1}$$

where: x is vertex position after deformation, x_0 is the vertex position before deformation, N is vertex normal after deformation, N_0 is the vertex normal before deformation, T_0 is the transformation matrix for the first skeleton segment, T_1 is the transformation matrix for the second skeleton segment, and w is the weight assigned to the vertex. For the first segment, the weighting factor is w, and for the second $1 - w$.

From Equation A.1, we see that a vertex's final position and normal are determined by linear interpolation, as the weight varies from 0 to 1 according to how the vertex's position is related to the skeleton segments and the skeleton structure.

In our deformation method, we define the vertex weight based on two structures: two-link and multi- link structures. In a two link structure as shown in Figure A.13 (left), a joint

is linked to two skeleton segments. For the human model, most pairs of skeleton segments are two-link structures, such as, the upper arm-lower arm, lower arm-hand, thigh-calf, calf-foot, etc. However, a multi-link structure is also required as shown in Figure A.13 (right), for joints linked to more than two skeleton segments. Examples of such a linkage are the skeleton segments uppertrunk, left shoulder, right shoulder and neck, as shown in Figure A.6 (right).

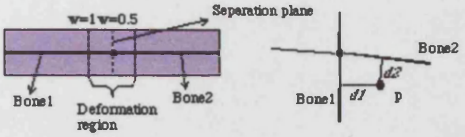


Figure A.13: Computing weights for different structures: *left:* two-link structure, *right:* multi-link structure (see following text for explanation of terms).

For the first structure, we divide the surface part corresponding to a skeleton into two regions: deformable region and undeformable region. The vertices within the undeformable region are far from a skeleton joint and their weight equals 1. The vertices within the deformable region are near the joint. Their weights are between 0.5 and 1.0. As shown in Figure A.13 (left), the weight of the vertex on the separating plane is 0.5, and the weight of the vertex on the plane separating the deformable region and undeformed region is 1.0. The value of the weight for a vertex within the deformable region can be computed according to the mapping parameter u of the vertex (defined in Figure A.11).

For the multi-layer structure, the computation of weights is more complicated. In some feature-based image morphing algorithms [BN92] and inverse distance weighted interpolation algorithms [She68], the distance between two geometric elements is usually considered as a weighting factor. In our method, we also use the distance from the vertex to the skeleton segment to compute the vertex's weight. To reduce the computational cost of deformation, however, we only use the two skeleton segments which have the most influence on the vertex to compute the weight. As shown in Figure A.13 (right), the bones that have most influence are $bone1$ and $bone2$, so we use them to compute the weight of the vertex. Thus, if the distances from the vertex to $bone1$ and $bone2$ are $d1$ and $d2$, respectively, the weight $w1$ of the vertex related to $bone1$ and the weight $w2$ related to $bone2$ is calculated as

$$w1 = \frac{d_2^2}{d_1^2 + d_2^2} \; and \; w2 = \frac{d_1^2}{d_1^2 + d_2^2} \tag{A.2}$$

Vertices of two link structures in undeformable regions can be mapped to a single bone, as

their weights are completely determined by the u parameter of its assigned bone. With multi-link structures vertices are mapped to the two closest bones. Mapping vertices to more than one bone in deformable regions of two link structures and multi-link structures, allows one to use the mapping parameters such as the angle to each bone, to avoid problems that may arise in some kinds of animation [Web00].

A.5 Results

In our human animation system, we apply BVH motion sequences to animate the scanned human model, as described in Section A.3.5. Figure A.14 shows some snapshots from results of such animation. In particular it shows the benefits of retaining more vertices around joint areas. In this figure, we used a uniformly reduced model in the top row (from Chapter 4), and non-uniformly reduced model in the bottom row (from Chapter 5). Both models had 5,000 triangles and approximately 2,531 vertices. Figure A.15 shows results that we were able to generate very quickly by applying our method to a variety of human models. We obtain the animation at about 100 frames/second on a PC with a 650 MHz Pentium III CPU, 128 MB of memory and a GeForce2 graphics card. Results with an even larger number of models can be seen in Figure A.16.

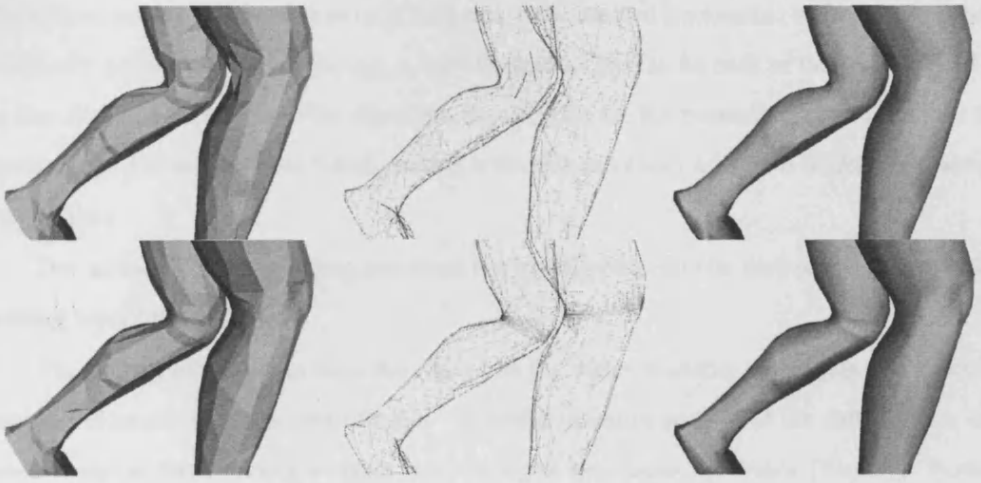


Figure A.14: *LoD for animation*; results from our non-uniformly simplified body (bottom) versus the uniformly simplified body (top) under deformation.

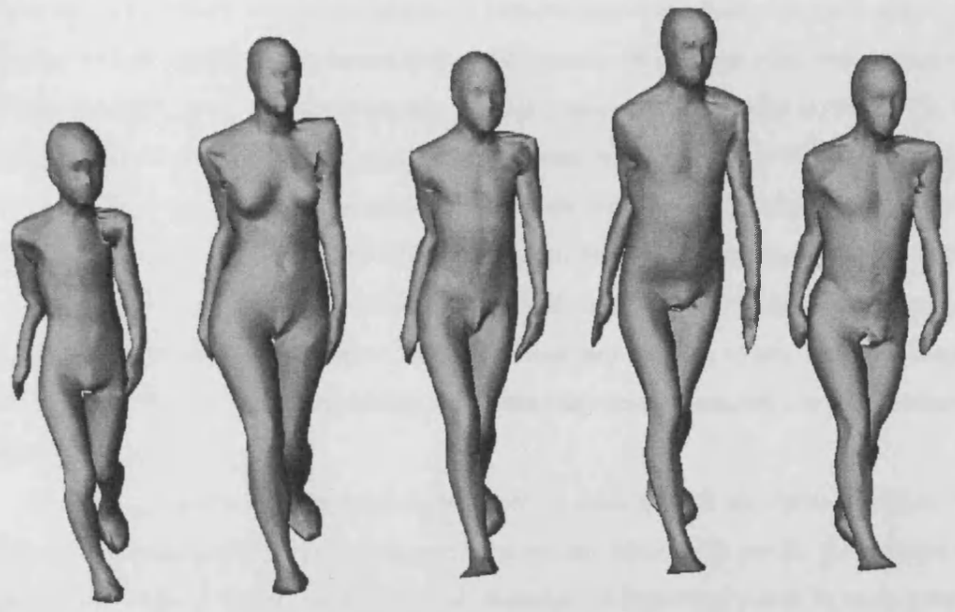


Figure A.15: Motion capture data applied to different models.

A.6 Conclusions

In this appendix, the skeleton of a scanned human model is built automatically by making use of automatically extracted key landmarks. This method is very useful for generating articulated models without the interaction of the user. The results of our automatic system shown in Figure A.16 would not have been possible if the techniques were not automatic at every stage. The system however is flexible to cope with manually selected landmarks. We selected slightly differently positioned landmarks (e.g. a knee landmark being at the back of the knee, instead of on the side of the knee), and the algorithm ensured that the corresponding nearest joint in the approximate medial axis was found, making it flexible and easily adaptable to different motion capture data.

Our surface to bone mapping technique has equally proved to be very robust in improving existing modelling paradigms.

The human deformation algorithm based on the vertex blending techniques can generate smooth deformation and is very efficient. To further improve realism of the deformation, one could compute the blending weights with strategies developed by Weber [Web00]. Further reducing the number of vertices in the model as described in [OB01] should allow for many avatars to be animated, if necessary simultaneously.

We showed the benefits of using our vertex classification system to retain more vertices in

joint areas. In this thesis we have not addressed collision detection which is of course necessary whenever there is significant movement of the model/avatar. We envisage a two step strategy for collision detection in our animation system. We plan to use a method similar to [BCH+95], for the first stage of collision detection, which uses a cascade of tests, based on the intersection test of bounding volumes quickly to eliminate most of the object pairs from detailed considerations. If the higher-level intersection test of a pair of cylinders has an intersection, we will go further to detect accurate collisions. We only need to check collisions for the geometric elements that are inside the intersection space. For this second step we plan to use vertex-to-triangle collision tests [Web00]. Finally we plan to use spatial and temporal coherence to help predicting collisions before deformation.

In addition, further work remains to be done. A more general algorithm should be developed for automatically generating the skeleton for any deformable model, for example by applying the ideas of segmentation based on detection of branching points to more generic models. Second, an efficient deformation algorithm that can reflect the muscle deformation should be developed. In our deformation algorithm using the vertex blending technique, if we can improve the weighting function of the vertices by considering the influence of mapped muscle deformation, the results will be more realistic and equally fast, since the muscles could be modelled with the same three position parameters defined in our current mapping.

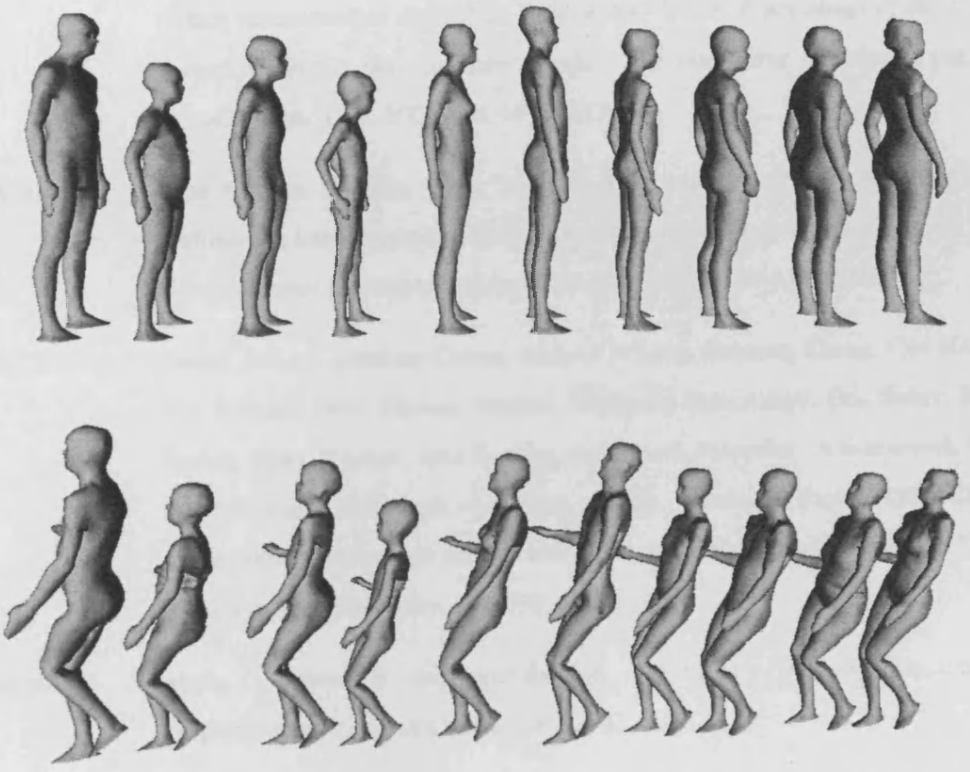


Figure A.16: Automatic skeletons on 10 simplified body scans, and animation.

Bibliography

[ABK98] Nina Amenta, Marshall Bern, and Manolis Kamvysselis. A new voronoi-based surface reconstruction algorithm. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 415–421, New York, NY, USA, 1998. ACM.

[ACDL02] Nina Amenta, Sunghee Choi, Tamal K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *International Journal of Computational Geometry and Applications*, 12(1-2):125–141, 2002.

[ACW+98] Daniel Aliaga, Jonathan Cohen, Andrew Wilson, Hansong Zhang, Carl Erikson, Kenneth Hoff, Thomas Hudson, Wolfgang Stuerzlinger, Eric Baker, Rui Bastos, Mary Whitton, Fred Brooks, and Dinesh Manocha. A framework for the real-time walkthrough of massive models. Technical Report TR98-013, Department of Computer Science and Engineering Faculty of Science and Engineering York University, 26, 1998.

[app06] apple. apple g5 chip specification. `http://www.apple.com/g5processor/architecture.html`, 2006.

[Ass93] Henry Dreyfuss Associates. *The Measure of Man and Woman Human factors in design.* Henry Dreyfuss Associates, 1993. ISBN –471-09955-4.

[Bad90] Didier Badouel. An efficient ray-polygon intersection. pages 390–393, 1990. includes code.

[Bau74] Bruce G. Baumgart. *Geometric modelling for computer vision.* PhD thesis, Stanford University, Department of Computer Science, California, U.S.A., 1974.

[BCH+95] R. Boulic, T. Capin, Z. Huang, P. Kalra, B. Lintermann, N. Magnenat-Thalmann, L. Moccozet, T. Molet, I. Pandzic, K. Saar, A. Schmitt, J. Shen, and

D. Thalmann. The HUMANOID environment for interactive animation of multiple deformable human characters. *Computer Graphics Forum*, 14(3):C/337–C/348, September 1995.

[BDDV02] Bernard Buxton, Laura Dekker, I. Douros, and T. Vassilvev. Reconstruction and interpretation of 3d whole body surface images. In *Numerisation 3D - Scanning 2002*. Harbour, BP 80126-35801 DINARD CEDEX, April 2002.

[Bel95] Gavin Bell. ivnorm. 1995. webpage last accessed 12th January 2004.

[BGB+05] Louis Borgeat, Guy Godin, Francois Blais, Philippe Massicotte, and Christian Lahanier. Gold: interactive display of huge colored and textured models. *ACM Trans. Graph.*, 24(3):869–877, 2005.

[bio03] biovision. bvhformat. `http://www.biovision.com/bvh.html`, 2003.

[BN92] Thaddeus Beier and Shawn Neely. Feature-based image metamorphosis. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 35–42. ACM Press, 1992.

[Bou97] Paul Bourke. Determining whether a line segment intersects a 3 vertex facet. `http://astronomy.swin.edu.au/~pbourke/geometry/linefacet/`, 1997.

[BPZ99] Chandrajit L. Bajaj, Valerio Pascucci, and Guozhong Zhuang. Single resolution compression of arbitrary triangular meshes with properties. In *Data Compression Conference*, pages 247–256, 1999.

[BS02] Jeffrey Bolz and Peter Schröder. Rapid evaluation of catmull-clark subdivision surfaces. In *Proceedings of the Web3D 2002 Symposium (WEB3D-02)*, pages 11–18, February 2002.

[BSGM02] W. Baxter, A. Sud, N. Govindaraju, and D. Manocha. Gigawalk: interactive walkthrough of complex environments. In *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, pages 203–214. Eurographics Association, 2002.

[BWK02] Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. Efficient high quality rendering of point sampled geometry. In *EGRW '02: Proceedings of the 13th*

Eurographics workshop on Rendering, pages 53–64, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Cat66] R. B. Catell. The scree test for the number of factors. *Multivariate Behavioural Research*, 1:245–276, 1966.

[CCMS96] Andrea Ciampalini, Paolo Cignoni, Claudio Montani, and Roberto Scopigno. Multiresolution decimation based on global error. Technical-report, CNR - Istituto di Elaborazione della Informazione (Pisa), 1996. Technical Report 1996-B4-026-10.

[CGG+04] Paolo Cignoni, Fabio Ganovelli, Enrico Gobbetti, Fabio Marton, Federico Ponchio, and Roberto Scopigno. Adaptive TetraPuzzles – efficient out-of-core construction and visualization of gigantic polygonal models. In *SIGGRAPH*, volume 23, New York, NY, USA, August 2004. ACM Press. Proc. SIGGRAPH 2004.

[Cho97] Mike M. Chow. Optimized geometry compression for real-time rendering. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 347–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.

[CHP89] J. E. Chadwick, D. R. Haumann, and R. E. Parent. Layered construction for deformable animated characters. In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, pages 243–252. ACM Press, 1989.

[Cig] Paolo Cignoni. Haussdorf.

[CKS02a] W. Correa, J. Klosowski, and C. Silva. iwalk: Interactive out-of-core rendering of large models. Technical Report TR-653-02, Princeton University, 2002.

[CKS02b] Wagner T. Correa, James T. Klosowski, and Cláudio T. Silva. Out-of-core sort-first parallel rendering for cluster-based tiled displays. In *EGPGV '02: Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization*, pages 89–96, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[Cla76] James H. Clark. Hierarchical geometric models for visible surface algorithms. *Communications of the ACM*, 19(10):547–554, 1976.

[CM02] P. Chopra and J. Meyer. Tetfusion: An algorithm for rapid tetrahedral mesh simplification. In *Proc. IEEE Visualization*, pages 133–140, 2002.

[COCSD03] Daniel Cohen-Or, Yiorgos L. Chrysanthou, Cláudio T. Silva, and Durand Durand. A survey of visibility for walkthrough applications. *IEEE Transactions on Visualization and Computer Graphics*, 09(3):412–431, 2003.

[Coh99] Jonathan David Cohen. *Appearance-Preserving Simplification of Polygonal Models*. PhD thesis, University of North Carolina, Chapel Hill, U.S.A., 1999.

[COL96] Daniel Cohen-Or and Yishay Levanoni. Temporal continuity of levels of detail in delaunay triangulated terrain. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 37–42, 1996.

[COM98] Jonathan Cohen, Marc Olano, and Dinesh Manocha. Appearance-preserving simplification. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 115–122, New York, NY, USA, 1998. ACM Press.

[CRMS03] Paolo Cignoni, Claudio Rocchini, Claudio Montani, and Roberto Scopigno. External memory management and simplification of huge meshes. *IEEE Transactions on Visualization and Computer Graphics*, 9(4):525–537, 2003.

[CRS98] P. Cignoni, C. Rocchini, and R. Scopigno. Metro: Measuring error on simplified surfaces. *Computer Graphics Forum*, 17(2):167–174, 1998.

[CVM+96] Jonathan Cohen, Amitabh Varshney, Dinesh Manocha, Greg Turk, Hans Weber, Pankaj Agarwal, Frederick Brooks, and William Wright. Simplification envelopes. *Computer Graphics*, 30(Annual Conference Series):119–128, 1996.

[CZ92] David T. Chen and David Zeltzer. Pump it up: computer animation of a biomechanically based model of muscle using the finite element method. In *Proceedings of the 19th annual conference on Computer graphics and interactive techniques*, pages 89–98. ACM Press, 1992.

[DDBT99] Laura Dekker, I. Douros, B. Buxton, and P. Treleaven. Building symbolic information for 3d human body modelling from range data. In *Second International Conference on 3-D Digital Imaging and Modelling, IEEE Computer Society*, pages 388–397, 1999.

[Dee95] Michael Deering. Geometry compression. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 13–20, New York, NY, USA, 1995. ACM Press.

[Dey02] T. Dey. Cocone homepage. `http://www.cis.ohio-state.edu/~tamaldey/cocone.html`, 2002.

[DG03] Tamal K. Dey and Samrat Goswami. Tight cocone: a water-tight surface reconstructor. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 127–134, New York, NY, USA, 2003. ACM.

[Die99] S. Dietrich. Vertex blending under directx 7 for the geforce256, technical presentations, 1999.

[DKT98] Tony DeRose, Michael Kass, and Tien Truong. Subdivision surfaces in character animation. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 85–94, New York, NY, USA, 1998. ACM Press.

[DKW+98] Laura Dekker, S. Khan, E. West, B. Buxton, and P. Treleaven. Models for understanding the 3d human body form. In *IEEE International Workshop on Model Based 3D Image Analysis*, pages 65–74, 1998.

[DLG90] Nira Dyn, David Levine, and John A. Gregory. A butterfly subdivision scheme for surface interpolation with tension control. *ACM Trans. Graph.*, 9(2):160–169, 1990.

[DR02] Christopher DeCoro and Pajarola Renato. XFastMesh: Fast view-dependent meshing from external memory. In Robert Moorhead, Markus Gross, and Kenneth I. Joy, editors, *Proceedings of the 13th IEEE Visualization 2002 Conference (VIS-02)*, pages 363–370, Piscataway, NJ, October 27– November 1 2002. IEEE Computer Society.

[DS98] D. Doo and M. Sabin. Behaviour of recursive division surfaces near extraordinary points. *Seminal graphics: poineering efforts that shaped the field*, pages 177–181, 1998.

[DZ91] Michael J. DeHaemer and Michael J. Zyda. Simplification of objects rendered by polygonal approximations. *Computers and Graphics*, 15(2):175–184, 1991.

[EDD+95] M. Eck, Tony DeRose, Tom Duchamp, Hugues Hoppe, Michael Lounsbery, and Werner Stuetzle. Multiresolution analysis of arbitrary meshes. *Computer Graphics*, 29(Annual Conference Series):173–182, 1995.

[EM00] Carl Erikson and Dinesh Manocha. Hierarchical levels of detail for fast display of large static and dynamic environments. Technical Report TR00-012, Chapel Hill, 4, 2000.

[ESC00] Jihad El-Sana and Yi-Jen Chiang. External memory view-dependent simplification. *Computer Graphics Forum*, 19(3):139–150, 2000.

[ESV98] J. El-Sana and A. Varshney. Topology simplification for polygonal virtual environments. *IEEE Transactions on Visualization and Computer Graphics*, 4(2):133–144, /1998.

[ESV99] J. El-Sana and A. Varshney. Generalized view-dependent simplification. *Computer Graphics Forum*, 18, No. 3:83 – 94, 1999.

[FCGW02] Guangzheng Fei, Kangying Cai, Baining Guo, and Enhua Wu. An adaptive sampling scheme for out-of-core simplification. *Comput. Graph. Forum*, 21(2):111–119, 2002.

[FGP98] P. Fua, A. Grün, and Apuzzo N. Thalmann D. Plänkers, R. Human body modelling and motion analysis from video sequences. In *International Symposium on Real-Time Imaging and Dynamic Analysis*, 1998.

[FMP98] Leila De Floriani, Paola Magillo, and Enrico Puppo. Efficient implementation of multi-triangulations. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 43–50, 1998.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. *Computer Graphics*, 27(Annual Conference Series):247–254, 1993.

[FSD99] Bruce Fischl, Martin I. Sereno, and Anders Dale. Cortical surface-based analysis ii: Inflation, flattening, and a surface-based coordinate system. *NeuroImage*, (9):2:195–207, 1999.

[FVFH90] Foley, VanDam, Feiner, and Hughes. *Computer Graphics Principles and Practice, 2nd Edition.* Addison Wesley, 1990.

[G.90] Fekete G. Rendering and managing spherical data with sphere quadtrees. *IEEE Visualization*, 14:176–186, 1990.

[Gar99] Michael Garland. *Quadric-based polygonal surface simplification /–Michael Garland.* PhD thesis, Carnegie Mellon University, 1999.

[GGS95] M. H. Gross, R. Gatti, and O. Staadt. Fast multiresolution surface meshing. In *Proceedings of the IEEE Visualization '95*, pages 135–142. IEEE Computer Society Press, 1995.

[GH98] Michael Garland and Paul S. Heckbert. Simplifying surfaces with color and texture using quadric error metrics. In David Ebert, Hans Hagen, and Holly Rushmeier, editors, *IEEE Visualization '98*, pages 263–270, 1998.

[Gil07] Duncan Fyfe Gillies. graphics slides. `http://www.doc.ic.ac.uk/~dfg/graphics/GraphicsSlides07.pdf`, 2007.

[GM05] Enrico Gobbetti and Fabio Marton. Far Voxels – a multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Transactions on Graphics*, 24(3):878–885, August 2005. Proc. SIGGRAPH 2005.

[GS97] Michael Garland and Heckbert Paul S. Surface simplification using quadric error metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[GS02] Michael Garland and E. Schaffer. A multiphase approach to efficient surface simplification. In *IEEE Visualization '02*, 2002.

[GSS99] Igor Guskov, Wim Sweldens, and Peter Schröder. Multiresolution signal processing for meshes. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 325–334, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[GTLH98] André Guéziec, Gabriel Taubin, Francis Lazarus, and William Horn. Simplicial maps for progressive transmission of polygonal surfaces. In *VRML '98: Proceedings of the third symposium on Virtual reality modeling language*, pages 25–31, New York, NY, USA, 1998. ACM Press.

[Gue96] Andre Gueziec. Surface simplification inside a tolerance volume. Technical report, Yorktown Heights, NY 10598, mar 1996. IBM Research Report RC 20440.

[GVSS00] Igor Guskov, Kiril Vidimce, Wim Sweldens, and Peter Schröeder. Normal meshes. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 95–102, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

[GZ05] Michael Garland and Yuan Zhou. Quadric-based simplification in any dimension. *ACM Trans. Graph.*, 24(2):209–239, 2005.

[HBG+99] A. Hilton, D. Beresford, T. Gentils, R. Smith, and W. Sun. Virtual people: Capturing human models to populate virtual worlds. In *IEEE International Conference on Computer Animation*, pages 174–185, 1999.

[HCK+99] Roger Hubbold, Jon Cook, Martin Keates, Simon Gibson, Toby Howard, Alan Murta, Adrian West, and Steve Pettifer. Gnu/maverik: a micro-kernel for large-scale virtual environments. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 66–73, New York, NY, USA, 1999. ACM.

[HDD+93] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, and Werner Stuetzle. Mesh optimization. *Computer Graphics*, 27(Annual Conference Series):19–26, 1993.

[HDG96] Roger J. Hubbold, Xiao Dongbo, and Simon Gibson. MAVERIK - the manchester virtual environment interface kernel. In M. Göbel, J. David, P. Slavik, and J. J. van Wijk, editors, *Virtual Environments and Scientific Visualization '96*, pages 11–20. Springer-Verlag Wien, 1996.

[Hel01] Martin Held. FIST: Fast industrial-strength triangulation of polygons. *Algorithmica*, 30(4):563–596, 2001.

[HH93] Paul Hinker and Charles Hansen. Geometric optimization. In *Proc. Visualization '93*, pages 189–195, San Jose, CA, October 1993.

[HHK+95] Taosong He, L. Hong, A. Kaufman, A. Varshney, , and S. Wang. Voxel-based object simplification. In *Proc. Visualization '95*. IEEE Comput. Soc. Press, 1995.

[HHVW96] Taosong He, Lichan Hong, Amitabh Varshney, and Sidney W. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, 1996.

[HK00] Roger J. Hubbold and M. J. Keates. Real-time simulation of a stretcher evacuation in a large-scale virtual environment. *Computer Graphics Forum*, 19(2):123–134, 2000.

[Hof89] C. M. Hoffmann. *Geometric and Solid Modeling*. Morgan Kaufmann, 1989.

[Hop96] Hugues Hoppe. Progressive meshes. *Computer Graphics*, 30(Annual Conference Series):99–108, 1996.

[Hop97] Hugues Hoppe. View-dependent refinement of progressive meshes. *Computer Graphics*, 31(Annual Conference Series):189–198, August 1997.

[Hop98] Hugues Hoppe. Efficient implementation of progressive meshes. *Computers and Graphics*, 22(1):27–36, 1998.

[Hop99] Hugues Hoppe. New quadric metric for simplifying meshes with appearance attributes. In David Ebert, Markus Gross, and Bernd Hamann, editors, *IEEE Visualization '99*, pages 59–66, San Francisco, 1999. IEEE.

[Hor95] Chiyoharu Horiguchi. Sensors that detect shape. *Advanced Automation Technology*, 7(5):210–216, 1995.

[Hor98] Chiyoharu Horiguchi. Bl (body line) scanner. *International Archives of Photogrammetry and Remote Sensing*, 32(5):421–429, 1998.

[IG03] Martin Isenburg and Stefan Gumhold. Out-of-core compression for gigantic polygonal meshes. In *Siggraph 2003, Computer Graphics Proceedings*, 2003.

[IL05] Martin Isenburg and Peter Lindstrom. Streaming meshes. In *IEEE Visualization*, page 30, 2005.

[KCK04] Deok-Soo Kim, Youngsong Cho, and Hyun Kim. Normal vector compression of 3d mesh model based on clustering and relative indexing. *Future Gener. Comput. Syst.*, 20(8):1241–1250, 2004.

[KG03] Youngihn Kho and Michael Garland. User-guided simplification. In *ACM Symposium on Interactive 3D Graphics*, pages 123–126, april 2003.

[KGC+96] Rolf M. Koch, Markus H. Gross, Friedrich R. Carls, Daniel F. von Büren, George Fankhauser, and Yoav I. H. Parish. Simulating facial surgery using finite element models. *Computer Graphics*, 30(Annual Conference Series):421–428, 1996.

[KMTM+98] Prem Kalra, Nadia Magnenat-Thalmann, Laurent Moccozet, Gael Sannier, Amaury Aubel, and Daniel Thalmann. Real-time animation of realistic virtual humans. *IEEE Computer Graphics and Applications*, 18(5):42–57, /1998.

[Knu73] D. E. Knuth. *Sorting and Searching, volume 3 of the Art of Computer Programming*. Addison-Wesley, 1973.

[Kob97] L. Kobbelt. Discrete fairing. In *Proceedings of the 7th IMA Conf. on the Mathematics of Surfaces*, pages 101–130, 1997.

[KS01] James T. Klosowski and Cláudio T. Silva. Efficient conservative visibility culling using the prioritized-layered projection algorithm. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):365–379, 2001.

[KSS98] R. Klein, A. Schilling, and W. Strasser. Illumination dependent refinement of multiresolution meshes. In *Proceedings of Computer Graphics International (CGI '98)*, pages 680–687, Los Alamitos, CA, 1998. IEEE Computer Society Press.

[KT96] Alan D. Kalvin and Russell H. Taylor. Superfaces: Polygonal mesh simplification with bounded error. *IEEE Computer Graphics and Applications*, 16(3):64–77, may 1996.

[Lan03] William Langdon. An implementation of the pseudo-random number generator park miller. `http://cs.ucl.ac.uk/genetic/gp-code/random-numbers`, 2003.

[LDS03] Haeyoung Lee, Mathieu Desbrun, and Peter Schröder. Progressive encoding of complex isosurfaces. *ACM Trans. Graph.*, 22(3):471–476, 2003.

[LE97] David Luebke and Carl Erikson. View-dependent simplification of arbitrary polygonal environments. *Computer Graphics*, 31(Annual Conference Series):199–208, 1997.

[LHNW00] David Luebke, Benjamin Hallen, Dale Newfield, and Benjamin Watson. Perceptually driven simplification using gaze-directed rendering. Technical Report CS-2000-04, University of Virginia, 2000.

[Lin00] Peter Lindstrom. Out-of-core simplification of large polygonal models. In Kurt Akeley, editor, *SIGGRAPH 20000, Computer Graphics Proceedings*, pages 259–262. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[Lis94] Dani Lischinski. *Incremental Delaunay Triangulation, Graphics Gems IV*, pages 47–59. Academic Press Professional, Inc., San Diego, CA, USA, 1994.

[LMP97] S. Levy, T. Münzner, and M. Phillips. Geomview, version 1.6.1. `http://www.geom.umn.edu/software/geomview/`, 1997.

[Loo87] Charles Loop. Smooth subdivision surfaces based on triangles. Master's thesis, University of Utah, 1987.

[Lou94] Michael Lounsbery. *Multiresolution Analysis for Surfaces of Arbitrary Topological Type*. PhD thesis, University of Washington, 1994.

[LPC+00] Marc Levoy, Kari Pulli, Brian Curless, Szymon Rusinkiewicz, David Koller, Lucas Pereira, Matt Ginzton, Sean Anderson, James Davis, Jeremy Ginsberg, Jonathan Shade, and Duane Fulk. The digital michelangelo project: 3D scanning of large statues. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 131–144. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[LS01a] Peter Lindstrom and Cláudio T. Silva. A memory insensitive technique for large model simplification. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 121–126, Washington, DC, USA, 2001. IEEE Computer Society.

[LS01b] Peter Lindstrom and Cláudio T. Silva. A memory insensitive technique for large model simplification. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 121–126, Washington, DC, USA, 2001. IEEE Computer Society.

[LSS+98] Aaron W. F. Lee, Wim Sweldens, Peter Schröder, Lawrence Cowsar, and David Dobkin. MAPS: Multiresolution adaptive parameterization of surfaces. *Computer Graphics*, 32(Annual Conference Series):95–104, 1998.

[LT99] Peter Lindstrom and Greg Turk. Evaluation of memoryless simplification. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):98–115, / 1999.

[LW01] Gong Li and Benjamin Watson. Semiautomatic simplification. In *Symposium on Interactive 3D Graphics*, pages 43–48, 2001.

[MMT97] Laurent Moccozet and Nadia Magnenat-Thalmann. Dirichlet free-form deformations and their application to hand simulation. In *Computer Animation, CA*, 1997.

[MT97] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *Journal of Graphics Tools: JGT*, 2(1):21–28, 1997.

[Neb00] J. C. Nebel. Soft tissue modelling from 3d scanned data. In *Deform 2000*, 2000.

[OB99] João Fradinho Oliveira and Bernard Buxton. Creating light-weight virtual humans for virtual environments. In *Eurographics'99, Short papers and Demos*, pages 15–18, September 1999.

[OB01] João Fradinho Oliveira and Bernard Buxton. Light weight virtual humans. In *Eurographics UK Chapter, 19th Annual Conference*, pages 45–52, April 2001.

[OB05] João Fradinho Oliveira and Bernard Buxton. An efficient octree for interactive large model visualization. Technical Report RN-05-13, Department of Computer Science, University College London, June, 13th 2005.

[OB06] João Fradinho Oliveira and Bernard Francis Buxton. Pnorms: platonic derived normals for error bound compression. In *VRST '06: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 324–333, New York, NY, USA, 2006. ACM.

[OS02] João Fradinho Oliveira and Anthony Steed. Determining orientation of laser scanned surfaces. In *SIACG, 1st Ibero-American Symposium in Computer Graphics*, pages 281–288, July 2002.

[OZSB03] João Fradinho Oliveira, Dongliang Zhang, Bernhard Spanlang, and Bernard Buxton. Animating scanned human models. *WSCG*, 11(2):362–369, 2003.

[PH97] Jovan Popović and Hugues Hoppe. Progressive simplicial complexes. *Computer Graphics*, 31(Annual Conference Series):217–224, August 1997.

[ply05] plyorient. plyorient. `http://www.cc.gatech.edu/projects/large_models/files/ply.tar.gz`, 2005.

[PTVF92] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *NUMERICAL RECIPES in C, The Art of Scientific Computing, 2nd Edition.* Cambridge Univeersity Press, 1992.

[Pup98] Enrico Puppo. Variable resolution triangulations. *Computational Geometry*, 11(3-4):219–238, 1998.

[R307] 3D Studio Max R3. 3d studio max r3. `http://www.discreet.com/3dsmax/`, 2007.

[RB93] Jarek Rossignac and Paul Borrel. Multi-resolution 3d approximations for rendering complex scenes. In B. Falcidieno and T. L. Kunii, editors, *Geometric Modeling in Computer Graphics*, pages 455–465. Springer-Verlag, 1993.

[RCB98] Charles Rose, Michael F. Cohen, and Bobby Bodenheimer. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications*, 18(5):32–41, /1998.

[RCC+01] C. Rocchini, P. Cignoni, Montani C., P. Pingi, R. Scopigno, R. Fontana, M. Greco, E. Pampaloni, L. Pezzati, M. Cygielman, R. Gianchetti, G. Gori, M Miccio, and R. Pecchioli. 3d scanning the minerva of arezzo. In *ICHIM'01*, pages 265–272, 2001.

[Red95] Martin Reddy. Musings on volumetric level of detail for virtual environments. *Virtual Reality: Research, Develpment and Application*, 1(1):49–56, 1995.

[Red97] Martin Reddy. *Perceptually Modulated Level of Detail for Virtual Environments.* Dissertation, University of Edinburgh, 1997.

[Rie07] Riegl. Riegl. `http://www.riegl.com/`, 2007.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In Kurt Akeley, editor, *SIGGRAPH 2000, Computer Graphics Proceedings*, pages 343–352. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[RR96] Remi Ronfard and Jarek Rossignac. Full-range approximation of triangulated polyhedra. *Computer Graphics Forum*, 15(3):C67–C76, C462, September 1996.

[RRH+02] David A. Randall, Todd D. Ringler, Ross P. Heikes, Phil Jones, and John Baumgardner. Climate modeling with spherical geodesic grids. 4(5):32–41, September/October 2002.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures.* Series in Computer Science. Addison-Wesley, Reading, Massachusetts, U.S.A., reprinted with corrections edition, April 1990.

[Sch97] Dieter Schmalstieg. Lodestar: An octree-based level of detail generator for VRML. In Rikk Carey and Paul Strauss, editors, *VRML 97: Second Symposium on the Virtual Reality Modeling Language*, New York City, NY, 1997. ACM Press.

[Sed90] Robert Sedgewick. *Algorithms in C.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[SG01] Eric Shaffer and Michael Garland. Efficient adaptive simplification of massive meshes. In Thomas Ertl, Ken Joy, and Amitabh Varshney, editors, *Proceedings Visualization 2001*, pages 127–134. IEEE Computer Society Technical Committee on Visualization and Graphics Executive Committee, 2001.

[She68] Donald S. Shepard. A two-dimensional interpolation function for irregularly spaced data. In *Proceedings of the 1968 ACM National Conference, New York*, pages 517–524, 1968.

[She06] C. K. Shene. cs3621: Introduction to computing with geometry notes,the euler-poincaré formula. `http://www.cs.mtu.edu/~shene/COURSES/cs3621/NOTES/model/euler.html`, 2006.

[SHSI01] W. Sun, A. Hilton, R. Smith, and J. Illingworth. Layered animation of captured data. 17(8):457–474, 2001.

[SL02] Jos Stam and Charles Loop. Quad/triangle subdivision. *submitted to Computer Graphics forum*, 2002.

[Slo81] Sloane. Tables of sphere packings and spherical codes. *IEEE Trans. Information Theory*, (27):327–338, 1981.

[SPCM97] Ferdi Scheepers, Richard E. Parent, Wayne E. Carlson, and Stephen F. May. Anatomy-based modeling of the human musculature. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 163–172. ACM Press/Addison-Wesley Publishing Co., 1997.

[SRC00] P. P. Sloan, C. F. III. Rose, and M. F. Cohen. Shape and animation by example. Technical-Report MSR-TR-2000-79, Microsoft Research, 2000.

[SS95] G. Schaufler and W. Stürzlinger. Generating multiple levels of detail from polygonal geometry models. In M. Göbel, editor, *Virtual Environments '95 (Eurographics Workshop)*, pages 33–41. Springer-Verlag: Heidelberg, Germany, 1995.

[SSC01] Mel Slater, Anthony Steed, and Yiorgos Chrysanthou. *Computer Graphics and Virtual Environments: From Realism to Real - Time*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[ST95] Jianhua Shen and Daniel Thalmann. Interactive shape design using metaballs and splines. In *Implicit Surfaces'95*, pages 187–196, Grenoble, France, April 1995. Proceedings of the first international workshop on Implicit Surfaces.

[Sta00] J. Stam. On subdivision schemes generalizing uniform b-spline surfaces of arbitrary degree. 2000.

[Ste98] Paul Steed. The art of low-polygonal modeling. `http://gdmag.com`, June 1998.

[Str88] Gilbert Strang. *Linear Algebra and its Applications*. Harcourt Brace Jovanovich College Publishers, Orlando, FL, USA, 3rd edition, 1988.

[Sun01] Dan Sunday. Intersections of rays and segments with triangles in 3d. `http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm`, 2001.

[Sun05] Sun. Sun microsystems dictionary. `http://docs.sun.com/db/doc/805-4368/6j450e60r?a=view`, 2005.

[SZL92] William J. Schroeder, Jonathan A. Zarge, and William E. Lorensen. Decimation of triangle meshes. *Computer Graphics*, 26(2):65–70, 1992.

[TGHL98] Gabriel Taubin, Andre Gueziec, William Horn, and Francis Lazarus. Progressive forest split compression. *Computer Graphics*, 32:123–132, 1998.

[THRL98] G. Taubin, W. Horn, J. Rossignac, and F. Lazarus. Geometry coding and vrml. In *Proceedings of the IEEE, Special issue on Multimedia Signal Processing*, volume 86, pages 1228–1243, June 1998.

[TLC02] Franco Tecchia, Celine Loscos, and Yiorgos Chrysanthou. Image-based crowd rendering. *IEEE Comput. Graph. Appl.*, 22(2):36–43, 2002.

[TR98] Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Transactions on Graphics*, 17(2):84–115, 1998.

[TS91] Seth J. Teller and Carlo H. Séquin. Visibility preprocessing for interactive walkthroughs. *Computer Graphics*, 25(4):61–68, 1991.

[TS96] D. Thalmann and Chauvineau E. Shen, J. Fast realistic human body deformations for animation and vr applications. In *Computer Graphics International*, pages 166–174. IEEE Computer Society Press, 1996.

[TT98] Marek Teichmann and Seth Teller. Assisted articulation of closed polygonal models. In *ACM SIGGRAPH 98 Conference abstracts and applications*, page 254. ACM Press, 1998.

[Tur92] Greg Turk. Re-tiling polygonal surfaces. *Computer Graphics*, 26(2):55–64, 1992.

[vRLJHK05] B. von Rymon-Lipinski, T. Jansen, N. Hanssen, and E. Keeve. Interactive visualization of large point isosurfaces using gpu-based decompression. In *Proceedings of the IEEE/Eurographics Symposium on Point-Based Graphics*, pages 55–64. ACM Press, 2005.

[WC07] formerly VRML Consortium Web3D Consortium. Virtual reality modeling language. `http://www.web3d.org/`, 2007.

[WDS99] Mason Woo, Davis, and Mary Beth Sheridan. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Web00] J. Weber. Run-time skin deformation. *Computer Game Developers Conference*, 2000.

[WEI06] E. W. WEISSTEIN. Platonic solid, frommathworld– a wolfram web resource. `http://mathworld.wolfram.com/PlatonicSolid.html`, 2006.

[WSNR96] B. Watson, V. Spaulding, Walker N., and W. Ribarsky. Evaluation of the effects of frame time variation on vr task performance. In *IEEE VRAIS*, pages 38–52, 1996.

[WWHW96] Benjamin Watson, Neff Walker, Larry F. Hodges, and Aileen Worden. Effectiveness of peripheral level of detail degradation when used with head-mounted displays. Technical Report 96-04, Graphics, Visualization & Usability Center, Georgia Institute of Technology, 1996.

[XV96] Julie C. Xia and Amitabh Varshney. Dynamic view-dependent simplification for polygonal models. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 335–344. IEEE, 1996.

[YSGM04] Sung-Eui Yoon, Brian Salomon, Russell Gayle, and Dinesh Manocha. Quick-VDR: Interactive view-dependent rendering of massive models. Technical Report TR04-011, Department of Computer Science, University of North Carolina - Chapel Hill, April 12 2004. Mon, 12 Apr 2004 17:34:34 UTC.

[ZM02] S. Zelinka and Garland M. Permission grids: Practical error-bounded simplification. *ACM Transactions on Graphics*, 21(2):207–229, 2002.

[ZMHKEH97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and III Kenneth E. Hoff. Visibility culling using hierarchical occlusion maps. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[ZSA04] Athanasios Zacharopoulos, Jan Sikora, and Simon Arridge. Parametric surface models in medical imaging. *Institute of Physics and Engineering in Medecine 04*, 10, 2004.

[ZZL03] Xu ZY, Tang ZS, and Tang L. An efficient rejection test for ray/triangle mesh intersection. *Journal of Software*, 14(10):1787 1795, 2003.

Index

UNIVERSITY OF LONDON

SENATE HOUSE. MALET STREET, LONDON, WCIE 7HU

REPRODUCTION OF THESES

A thesis which is accepted by the University for the award of a Research Degree is placed in the Library of the College and in the University of London Library. The copyright of the thesis is retained by the author.

As you are about to submit a thesis for a Research Degree, you are required to sign the declaration below. This declaration is separate from any which may be made under arrangements with the College at which you have *pursued* your course (for internal candidates only). The declaration will be destroyed if your thesis is not approved by the examiners, being either rejected or referred for revision.

Academic Registrar

To be completed by the candidate

NAME IN FULL (please type surname in BLOCK CAPITALS)

JOÃO FERNANDO DOS SANTOS FRADINHO DUARTE DE OLIVEIRA

THESIS TITLE

VERTEX CLASSIFICATION FOR NON-UNIFORM GEOMETRY REDUCTION

DEGREE FOR WHICH THESIS IS PRESENTED Please select either... PhD

DATE OF AWARD OF DEGREE *(To be completed by the University):* _____

DECLARATION

1. I authorise that the thesis presented by me in *[2008] for examination for the MPhil/PhD Degree of the University of London shall, if a degree is awarded, be deposited in the library of the appropriate College and in the University of London Library and that, subject to the conditions set out below, my thesis be made available for public reference, inter-library loan and copying.

2. I authorise the College or University authorities as appropriate to supply a copy of the abstract of my thesis for inclusion in any published list of theses offered for higher degrees in British universities or in any supplement thereto, or for consultation in any central file of abstracts of such theses.

3. I authorise the College and the University of London Libraries, or their designated agents, to make a microform or digital copy of my thesis for the purposes of inter-library loan and the supply of copies.

4. I understand that before my thesis is made available for public reference, inter-library loan and copying, the following statement will have been included at the beginning of my thesis: The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

5. I authorise the College and/or the University of London to make a microform or digital copy of my thesis in due course as the archival copy for permanent retention in substitution for the original copy.

6. I warrant that this authorisation does not, to the best of my belief, infringe the rights of any third party.

7. I understand that in the event of my thesis being not approved by the examiners, this declaration would become void.

***Please state year by hand, using a pen.**

DATE 7TH DECEMBER 2007 SIGNATURE _____

Note: The University's Ordinances make provision for restriction of access to an MPhil/PhD thesis and/or the abstract but only in certain specified circumstances and for a maxiumum period of two years. If you wish to apply for such restriction, please enquire at your College about the conditions and procedures. External Students should enquire at the Research Degree Examinations Office, Room 261, Senate House.